NAVAL POSTGRADUATE SCHOOL

Monterey, California

K4111 3

THESIS

STATIC SCHEDULERS FOR EMBEDDED REAL-TIME SYSTEMS

by

Murat Kilic

December 1989

Thesis Advisor: Uno R. Kodres

Approved for public release; distribution is unlimited.

T247932

nclassified

ecurity classification of this page

REPORT DOCUMENTATION PAGE

a Report Security Classification Unclassified	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report
2b Declassification Downgrading Schedule	Approved for public release; distribution is unlimited.
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization Naval Postgraduate School — 6b Office Symbol *(if applicable)* 52	7a Name of Monitoring Organization Naval Postgraduate School
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000	7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000
8a Name of Funding Sponsoring Organization — 8b Office Symbol *(if applicable)*	9 Procurement Instrument Identification Number
8c Address *(city, state, and ZIP code)*	10 Source of Funding Numbers Program Element No — Project No — Task No — Work Unit Accession No

11 Title *(include security classification)* STATIC SCHEDULARS FOR EMBEDDED REAL-TIME SYSTEMS

12 Personal Author(s) Murat Kilic

13a Type of Report	13b Time Covered	14 Date of Report *(year, month, day)*	15 Page Count
Master's Thesis	From To	December 1989	159

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms *(continue on reverse if necessary and identify by block number)*
Field	Group	Subgroup	Static Schedulers, Single Processor Scheduling, Nonpreemtive Scheduling, Implementation of Static Schedulars

19 Abstract *(continue on reverse if necessary and identify by block number)*

Because of the need for having efficient scheduling algorithms in large scale real time systems, software engineers put a lot of effort on developing scheduling algorithms with high performance. But algorithms, developed upto now, are not perfect for all cases. At this stage, instead of having one scheduling algorithm in the system, more than one different algorithm which will try to find a feasible solution to the scheduling problem according to the initial properties of tasks would be very useful to reach a high performance scheduling for the system.

This report presents the effort to provide static schedulers for the Embedded Real-Time Systems with single processor using Ada programming language. The independent nonpreemptable algorithms used used in three static schedulers are run according to the timing constraints and precedence relationships of the critical operators extracted from high level source program. The final schedule guarantees that timing constraints for the critical jobs are met. The primary goal of this report is to support the Computer Aided Rapid Prototyping for Embedded Real-Time Systems so that we determine whether the system, as designed, meets the required timing specifications. Secondary goal is to demonstrate the significance of Ada as the implementation language and a modeling tool for a prototyping system.

0 Distribution Availability of Abstract ☒ unclassified unlimited ☐ same as report ☐ DTIC users	21 Abstract Security Classification Unclassified	
2a Name of Responsible Individual Uno R. Kodres	22b Telephone *(include Area code)* (408) 646-2197	22c Office Symbol Code 52Kr

D FORM 1473.84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — security classification of this page

Unclassified

Static Schedulers for Embedded Real-Time Systems

by

Murat Kilic
Lieutenant J. G., Turkish Navy
B.S., Turkish Naval Academy

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1989

[illegible]

ABSTRACT

Because of the need for having efficient scheduling algorithms in large scale real-time systems, software engineers put a lot of effort on developing scheduling algorithms with high performance. But neither algorithm developed upto now is perfect for all cases. At this stage, instead of having one scheduling algorithm in the system, more than one different algorithms which will try to find a feasible solution to the scheduling problem according to the initial properties of the tasks would be very useful to reach a high performance scheduling for the system.

This report represents the effort to provide static schedulers for the Embedded Real-Time Systems with single processor using the Ada programming language. The independent nonpreemptable algorithms used in these static schedulers are run according to the timing constraints and precedence relationships of the critical operators extracted from a high level source program. The final schedule guarantees that timing constraints for the critical jobs are met. The primary goal of this report is to support the Computer Aided Rapid Prototyping for Embedded Real-Time Systems so that we will determine whether the system, as designed, will meet the required timing specifications. Secondary goal is to demonstrate the significance of Ada as the implementation language.

T K6313
K41113
C. 1

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Large scale Real-Time Systems are important to both civilian and military operations. They are used in the control of modern systems, in air traffic control, in tele-communication systems, and in defense. In these systems, many tasks have explicit deadlines. This means that the task scheduling is an important component of the systems. In Hard Real-Time Systems, tasks have to be performed not only correctly, but also in a timely fashion. Otherwise, there might be some severe consequences.[Ref. 12: p.3]

The scheduling algorithm in a Hard Real-Time System can be either static or dynamic, and is used to determine whether a feasible execution schedule for a set of tasks exists so that the tasks' deadlines and resource requirements are satisfied, and generate a schedule if one exists [Ref. 10]. A static approach calculates schedules for tasks off-line and it requires the complete prior knowledge of tasks' characteristics. A dynamic approach determines schedules for tasks on the fly and allows tasks to be dynamically invoked. Although static approaches have low run-time cost, they are inflexible and can not adapt to a changing environment or to an environment whose behavior is not completely predictable. When new tasks are added to a static system, the schedule for the entire system must be recalculated, which is expensive in terms of time and money. In contrast, dynamic approaches involve higher run-time costs, but, because of the way they are designed, they are flexible and can easily adapt to changes in the environment.[Ref. 12: p. 3] In Hard Real-Time Systems, tasks are also distinguished as preemptable and nonpreemptable. A task is preemptable if its execution can be interrupted by other tasks and resumed afterwards. A task is nonpreemptable if it must run to completion once it starts.

To meet timing constraints, we must schedule software tasks according to well understood algorithms, so that the resultant timing behavior of the system is understandable, maintainable and predictable. The use of well understood Real-Time scheduling algorithms will also set the stage for eliminating many of the time dependant problems encountered in Real-Time Systems today, thereby avoiding some of the most difficult problems to debug, with a resultant increase in system reliability and with reduced system integration time and cost [Ref. 11].

B. THE STATIC SCHEDULER

If there exists a possible solution, the static scheduler builds a static schedule for the execution of a prototype, which is a sequence of tasks being developed from the Prototype System Description Language(PSDL) input specification for the prototype that obey some predefined properties, in our case these are timing constraints and precedence relationships. This schedule gives the precise execution order and timing of operators with hard real-time constraints in such a manner that all timing constraints are guaranteed to be met [Ref. 14].

Tasks are divided into two classes: time-critical and non time-critical. A task is time-critical if it has at least one timing constraint associated with it, otherwise it is non time-critical. Time critical tasks need more work to get a feasible schedule, therefore they are handled by static scheduler before running a prototype.

And an auxiliary scheduler, called dynamic scheduler, executes the time-critical task sequence generated by static scheduler and tries to allocate the non time-critical tasks obeying the precedence relationship for the free time slots of CPU. The importance of the static scheduler is that it obtains a sequence for the critical tasks off-line, thus avoiding execution time during run time.

C. OBJECTIVES

This thesis describes the application of the schedulers that use different scheduling algorithms to find feasible schedules for the real-time prototypes satisfying the critical timing constraints and precedence relationships among operators in the prototype.

D. ORGANIZATION

Chapter II describes the previous research done in general. It includes a discussion of Computer Aided Prototype System(CAPS) and Prototype System Description language(PSDL). This chapter also presents a survey of The Static Scheduling Algorithms for single processor environment. Chapter III outlines the analysis and programming decisions that were made during the implementation. The deviations from the earliest implementation are described in Chapter IV. Conclusions and recommendations for the future work will be presented in Chapter V.

II. PREVIOUS RESEARCH AND SURVEY OF STATIC SCHEDULING ALGORITHMS

A. PREVIOUS RESEARCH

The research previously done in static scheduler is associated with the Computer Aided Prototyping System(CAPS) and the Prototype System Description Language (PSDL). CAPS is a tool that is being designed to aid software designers in the rapid prototyping of large software systems. The original design of the Static Scheduler was described in [Ref. 19]. This design was further developed as The Conceptual Design for the Pioneer Prototype of the Static Scheduler as a part of the CAPS execution support system.[Ref.14] Then we see a pioneering effort to develop a static scheduler as a part of the CAPS execution support system, using the Ada®[1] programming language.[Ref. 13]. Thereafter a static scheduler was partly implemented in the Ada® programming language. [Ref. 6]

1. CAPS

The CAPS is a tool that's being designed for development of Hard Real-Time or Embedded Systems to speed up the design and implementation. CAPS process is an iterative approach to designing complex software systems. CAPS is the major system that requires more than one static scheduler.

[1] Ada® is a registered trademark of the United States Government, Ada Joint Program Office

The CAPS architecture contains the following elements:

- User Interface
- Prototyping System Description Language
- Rewrite Subsystem
- Software Design Management System
- Prototype Data Base and Software Base
- Execution Support System(ESS)

Detailed information about CAPS is contained in [Ref.31], [Ref. 16], [Ref. 13], and [Ref. 6] Figure 1 below graphically describes the major software tools of CAPS, and the Figure 2 on page 6 describes the architecture of CAPS.

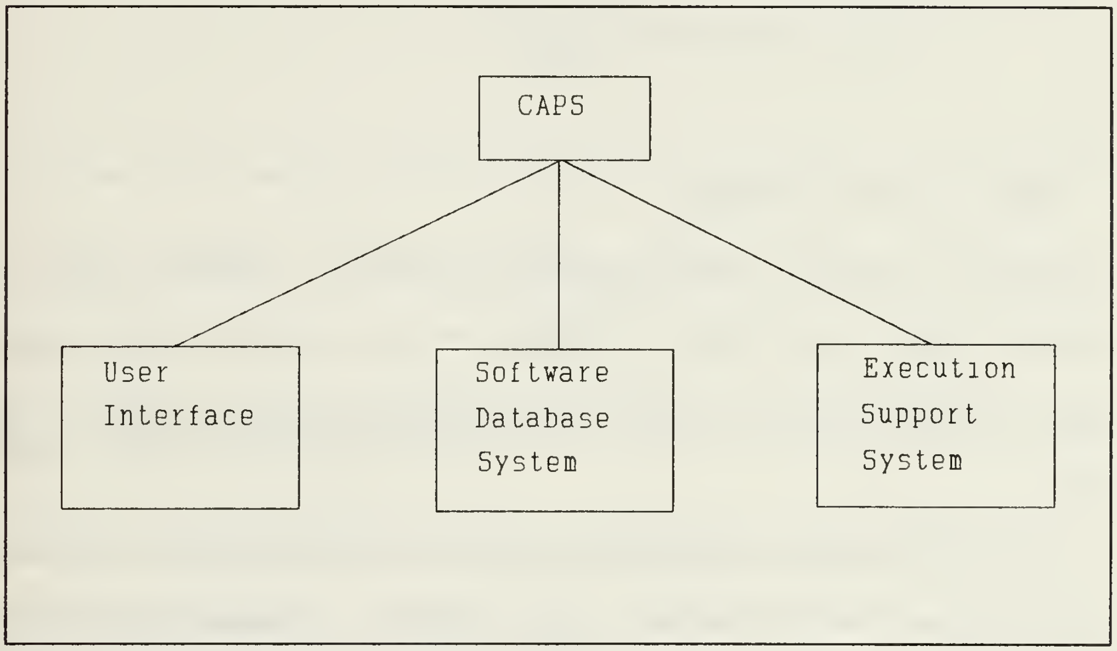


Figure 1 Major Software Tools of CAPS

CAPS makes use of specifications and reusable software components to automate the rapid prototyping methodology [Ref. 16: p. 66], which offers promising

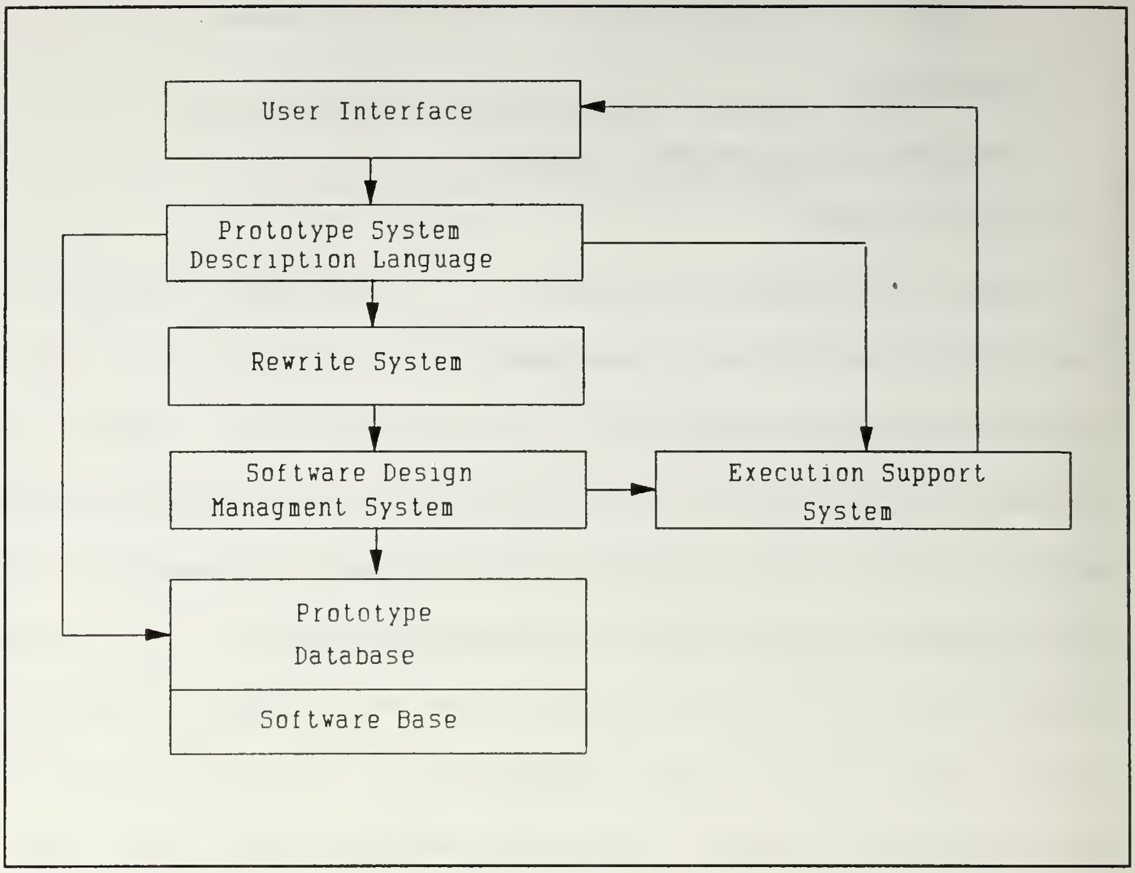


Figure 2 CAPS Architecture

advantages in improved software engineering productivity, increased reliability of the finished product, more realistic cost estimates based on identified system complexity, and a reduction in the total system design to implementation timelog [Ref. 15: pp. 11-12].

The Execution Support System is necessary for the execution and testing of the prototype. The ESS contains a Static Scheduler, a Translator, and a Dynamic Scheduler. [Ref. 32] The interfaces between these components are shown in figure 3 on page 7. The Translator translates the statements in the PSDL prototype into statements in an underlying programming language. The underlying programming

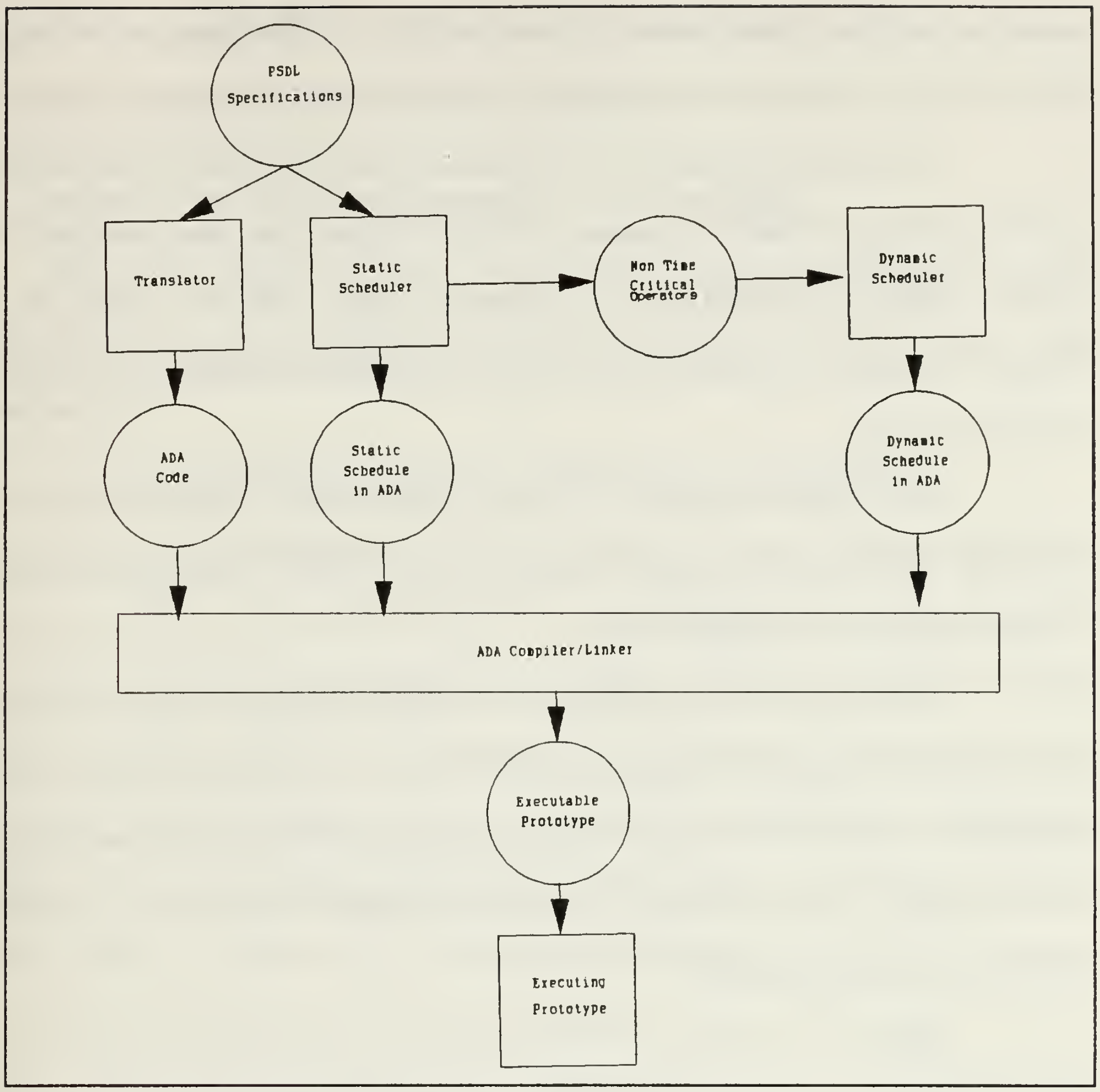


Figure 3 The Execution Support System

language for the CAPS is Ada®. The development of the translator is presented in Moffitt [Ref. 18].

Static Scheduler is a part of the ESS and attempts to find a static schedule for the operators in PSDL prototype with real-time constraints. An implementation guide for the Static Scheduler can be found in [Ref. 13]. The operators that do not have real time constraints are controlled by Dynamic Scheduler during run time.

The Dynamic Scheduler is a run time executive which controls the execution of the prototype, it schedules operators which do not have real time constraints, and provides facilities for debugging and gathering statistics. The first design for the Dynamic Scheduler is contained in [Ref. 28] and the latest changes can be found in Palazzo [Ref. 32].

The translator translates the PSDL code into Ada source code, the Static Scheduler extracts operator timing information from the PSDL source code and creates a static schedule in Ada source code.

The Static Scheduler provides the Dynamic Scheduler with the non time-critical operators. Dynamic Scheduler uses the "non_crits" text file to create a dynamic schedule in a Ada source code. And, the Ada source code from the Translater, the Static Scheduler, and the Dynamic Scheduler are compiled, linked and an executable Prototype is generated.[Ref.32]

2. PSDL

PSDL is a language designed for clarifying the requirements of complex real-time systems and for determining properties of proposed designs for such systems by means of prototype execution. The language was designed to simplify the description of such systems and to support a prototyping method that relies on a novel decomposition criterion. PSDL is also the basis for the CAPS that speeds up the prototyping process by exploiting reusable software components and providing execution support for high level constructs appropriate for describing large real-time systems in terms of an appropriate set of abstractions. [Ref. 17: p. 7]

B. SURVEY OF STATIC SCHEDULING ALGORITHMS

This section includes a survey of The Static Scheduling Algorithms for Hard Real-Time Systems, and presents an overview of previous work and discusses their characteristics.

1. THE FIXED PRIORITIES SCHEDULING ALGORITHM

In many conventional hard real-time systems, tasks are assigned with fixed priorities to reflect critical deadlines, and tasks are executed in an order determined by the priorities. During the testing period, the priorities are (usually manually) adjusted until the system implementer is convinced that the system works. Such approach can only work for relatively simple systems, because it is hard to determine a good priority assignment for a system with a large number of tasks by such a test-and-adjust method. Fixed priorities is a type of static scheduling. Once the priorities are fixed in a system, it is very hard and expensive to modify the priority assignment.[Ref. 27].

2. THE HARMONIC BLOCK WITH PRECEDENCE CONSTRAINTS SCHEDULING ALGORITHM

This scheduling algorithm is being used by the CAPS, a general description of the implementation is furnished above, and a Data Flow Diagram(DFD) is given in Figure 4 on page 10. After the first design efforts of this algorithm [Ref. 13] [Ref. 14] even though the data flow diagram didn't change since the first Architectural Design, some structural changes were made to the algorithm. Description below includes these final structural changes [Ref. 6].

The first component of the DFD, the "PSDL_Reader", reads and processes the PSDL prototype program. The output of this step is a file containing operators identifiers, timing information and link statements.

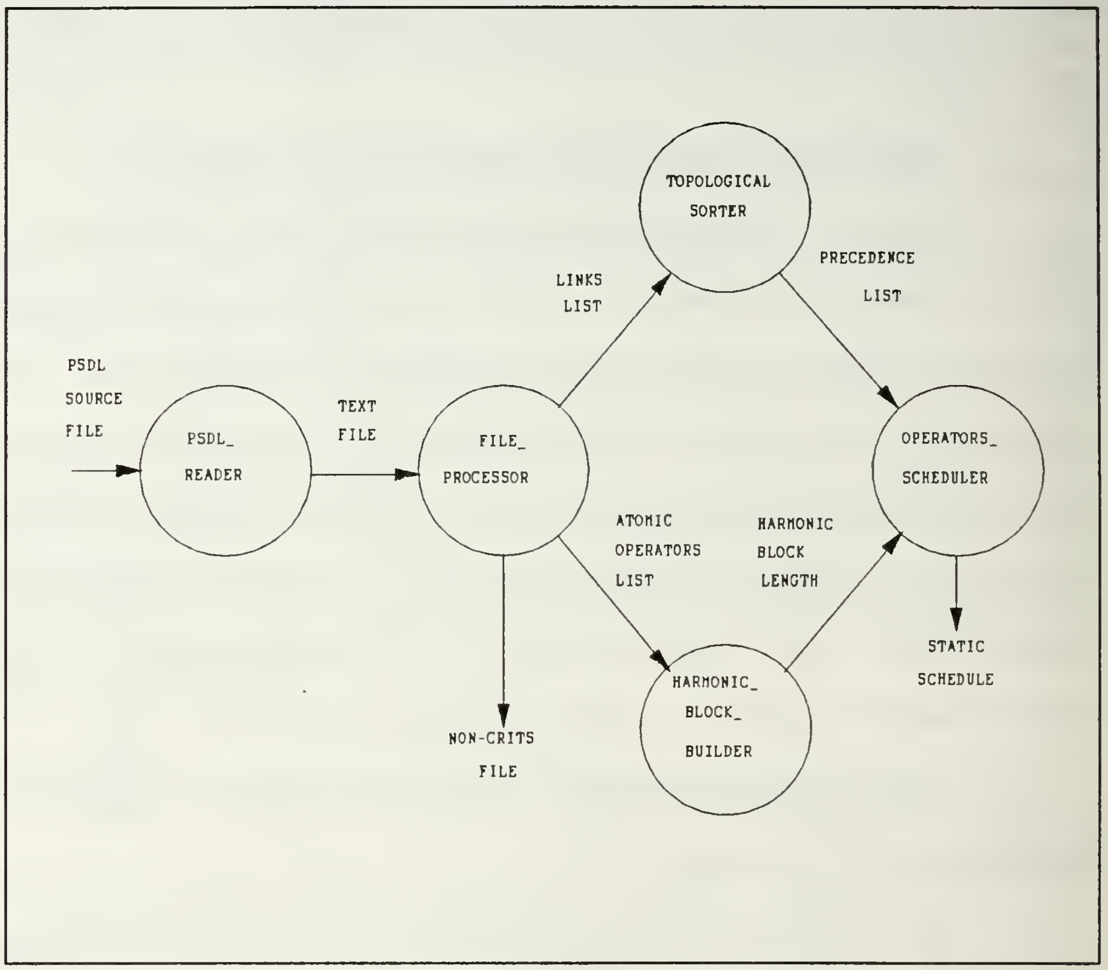


Figure 4 1" Level DFD

The second component is the "File_Processor", the file generated in the first step is analyzed and the data is divided into three parts based on its destination or if additional processing required. The "Non_Crits" file contains the names of all noncritical operators. The Atomic Operators list contains all critical operators identifiers and their associated timing constraints. The Links List contains the link statements which syntactically describe the PSDL implementation graphs. During this step some basic validity checks on the timing constraints are performed. If any of the checks fails, an exception is raised and an appropriate error message is submitted to the user.

The "Topological_Sorter" performs a topological sort of the link statements contained in the Links List. The requirements for a topological sort implies that the statements being sorted have natural continuity and connectedness. These properties define the execution precedence of the time critical operators regardless of whether the graphs are linear or acyclic. In an acyclic digraph, like on Figure 5, the decision to choose the "link a" first and the "link b" last is arbitrary in (b). The output from either sort is a precedence list of critical operators stipulating the exact order in which they must be executed. The linear sort will produce one precedence list while the acyclic sort can produce two or more precedence lists.

The second output of the "File_Processor", the Atomic Operators list, is the input to the "Harmonic_Block_Builder". An harmonic block is defined as a set of periodic operators where the periods of all its component operators are exact multiples of a calculated base period. Each harmonic block is treated as an independent scheduling problem. When multiprocessors are utilized, then one processor for harmonic block is necessary. The implementation being developed [Ref. 6] utilizes a single processor, therefore the final static schedule assumes that only one harmonic block is created. All the operators must be periodic, then all the sporadic operators are converted to their periodic equivalents. The periodicity helps to insure that execution

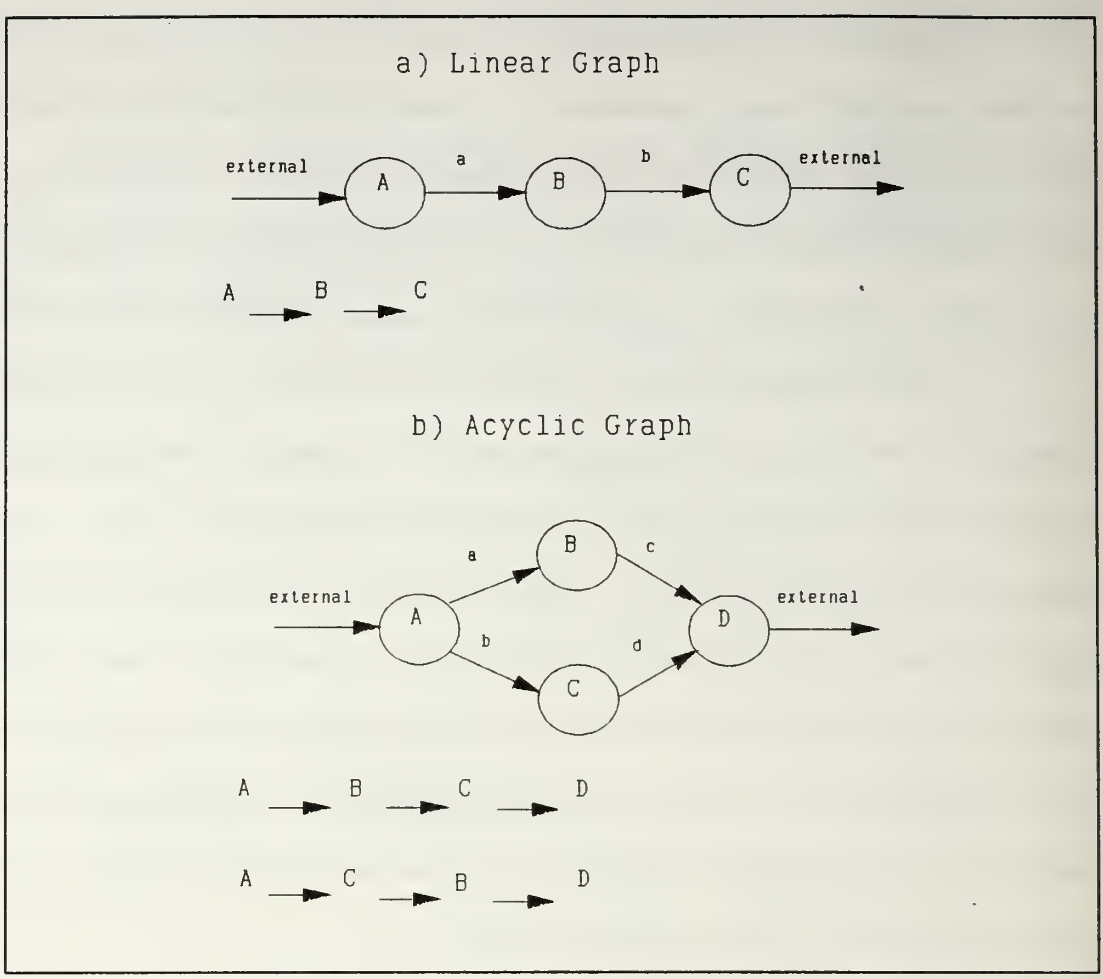


Figure 5 Linear and Acyclic Graphs

is completed between the beginning of a period and its deadline, which defaults to the end of the period.

In order to convert a sporadic operator into its equivalent periodic operator, the following parameters of the sporadic operator must be known :

- Maximum Execution Time (MET).
- Minimum Calling Period (MCP).
- Maximum Response Time (MRT).

Some rules must be obeyed by the parameters described above to obtain an equivalent periodic operator, the rules are the following :

- MET < MRT. This rules insures that (MRT - MET) produces a positive value.
- MCP < MRT. This condition is necessary, but not sufficient, to guarantee that an operator can fire at least once before a response is expected.
- MET < MCP. This restriction insures that the period calculated will conform to a single processor environment.

The periodic equivalent is then calculated as P = min (MCP, MRT - MET), the value of P must be greater than MET, in order for the operator to complete execution within the calculated period. As a last resort, setting P equal to MCP, is a worst case scheduling constraint.

After all the operators are in periodic form, they are sorted in ascending order based on the period values. A second preliminary step is to calculate the base block and its period for the sorted sequence of operators. The base period is defined as the greatest common divisor (GCD) of all the operators in one sequence that will be scheduled together.

The last preliminary step is to evaluate the length of time for the harmonic block. The actual harmonic block length is the least common multiple (LCM) of all the operators' period contained in the block. The harmonic block and its length are an integral part of the static schedule. This block represents an empty timeframe within which the operators will be allocated time slots for execution.

The outputs of the "Topological_Sorter" and the "Harmonic_Block_Builder" are used by the "Operators_Scheduler" in order to create a static schedule for the time critical operators. The resulting static schedule is a linear table giving the exact execution start time for each critical operator and the reserved MET within which each operator completes its execution.

This linear table is evaluated in two iterative steps. In the first step an execution time interval is allocated for each operator based on the equation INTERVAL = (current time, current time + MET). Next the process creates a firing interval for each operator during which the second iterative step must schedule the operator. The firing interval stipulates the lower and upper bound for the next possible start time for an operator based on its period. The second step, initially, uses the lower bound of each firing interval, when it schedules operators during subsequent iterations. The sequence of operators is allocated time slots according to the earliest lower bound first. Before an operator is allocated a time slot, this step verifies that :

- (current time + MET) = < harmonic block length.

This condition is applicable to every operator scheduled in that harmonic block. This step also calculates new firing intervals for each operator scheduled. Once all the operators are correctly scheduled within an entire harmonic block, a static schedule is available. All subsequent harmonic blocks are copies of the first.

A theoretical development and implementation guideline of this algorithm is available in the [Ref. 14] and [Ref. 13].

Part of the actual implementation of this algorithm and the analysis of its performance is described in the [Ref. 6].

3. THE EARLIEST START SCHEDULING ALGORITHM

This algorithm considers the scheduling of n tasks on a single processor. Each task becomes available for processing at time a_i, must be completed by time b_i, and requires d_i time units for processing.

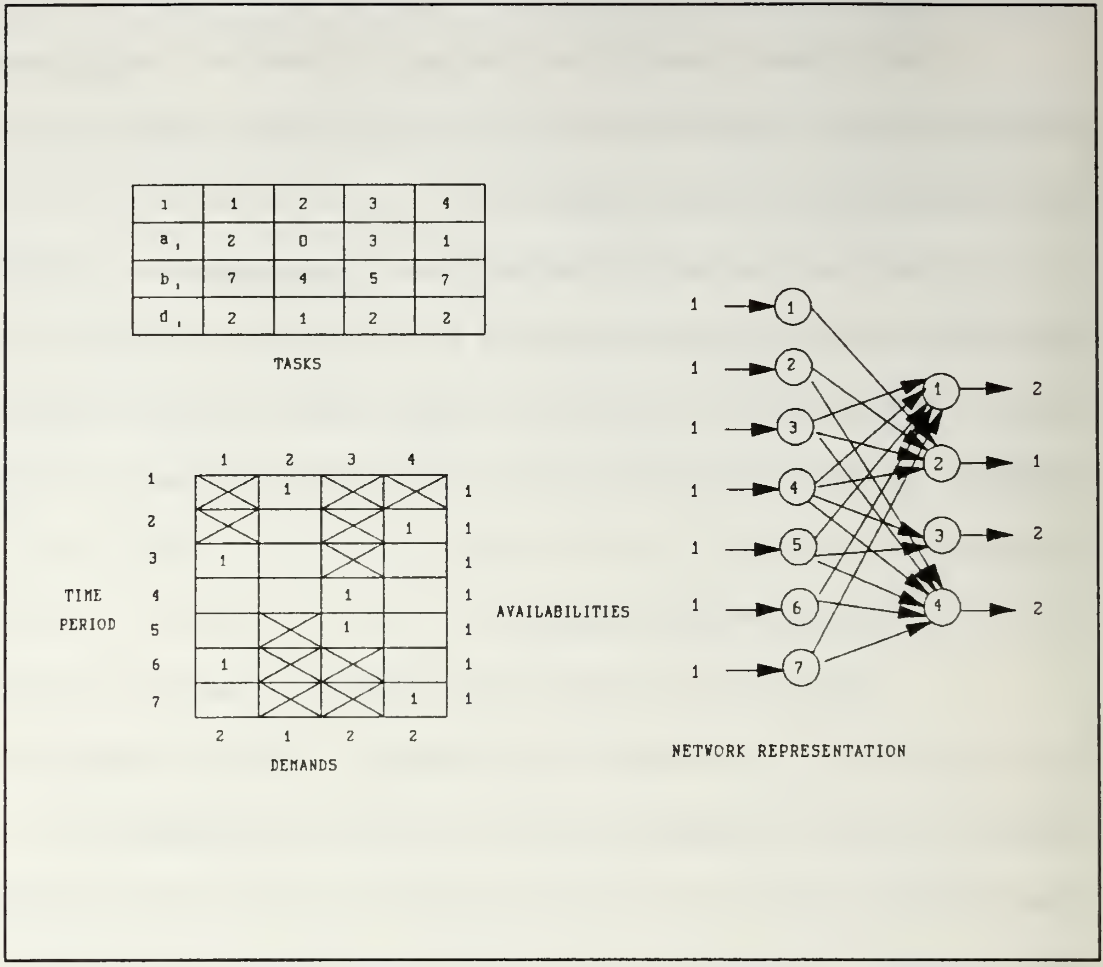
There are two versions of the criteria : one allows the job splitting (preemptable tasks), under this assumption it is only required to complete d^k_i (where $d^1_i+d^2_i+...+d^n_i=d_i$, and n is the total number of splits of the task i) units of processing between a_i and b_i; and the other version assumes that job splitting is not allowed (nonpreemptable tasks).

a. PREEMPTABLE VERSION

Consider the rectangular matrix that has a column for each job and a line for each unit of time available. There are $\max_i(b_i)$ lines and n columns. In this matrix it is necessary to distinguish between admissible and inadmissible cells. For job i the cell (i,j) is admissible, if $a_i<j=<b_i$, and inadmissible otherwise. The admissible cells correspond to the time where the task may be performed. The Figure 6 below shows an example.

Associated with each row there is an availability of one unit of time, and with each column a requirement of d_i. If the task i is being processed at time j, a 1 is placed in the admissible cell. This problem is equivalent to that of finding a set of 1's placed in admissible cells such that column sums satisfy the requirements d_i and each line contains at most one single 1.[Ref. 20: pp. 511-514]

This type of algorithm does not take into account any precedence constraints. In order to include the precedence constraints in this algorithm, it is



i	1	2	3	4
a_i	2	0	3	1
b_i	7	4	5	7
d_i	2	1	2	2

Figure 6 Example of Scheduling with Earliest Start Time (preemptable)

necessary to do some modifications. The modification can utilize some concept like the harmonic block discussed in the former algorithm and also include the constraints that a job j, that is preceded by i and k, is admissible only after $a_i<j=<b_i$ and i and k are already scheduled. The [Ref. 20: pp. 518-519] presents an implementation in FORTRAN to solve the case without precedence constraints. This type of algorithm is not taken into account for precedence constraints, and is not applicable to our case because it assumes that all the tasks are preemptable.

This algorithm is bounded by O(n) in time, and as most heuristic algorithms, does not guarantee that the solution (assuming that at least one is available for the problem) is found.

b. NONPREEMPTABLE VERSION

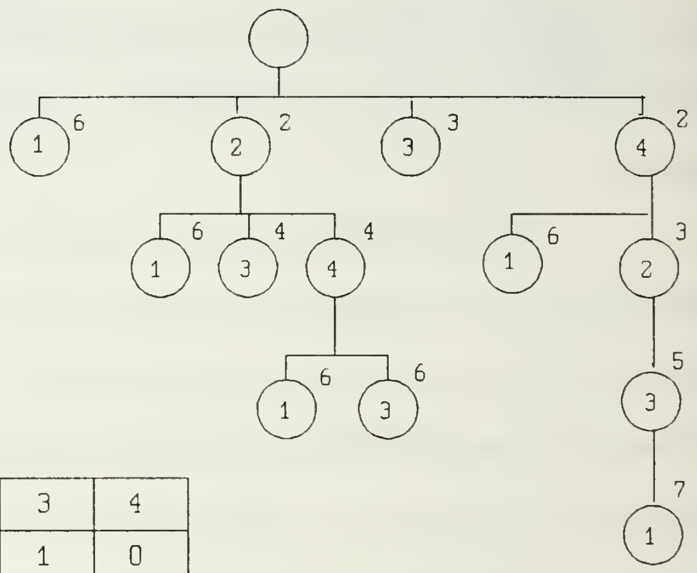
In this approach, also, the precedence constraints are not included in the analysis, but they may be easily taken into account during the construction of all the feasible sequences.

The main idea is to enumerate implicitly all the possible orderings by a branch, exclude and bound algorithm. During the branch all infeasible sequences due to violation of the due date are discarded (here it is possible to include the precedence constraints).

All the possible sequences are enumerated by a tree type construction. From the initial node we branch to n new nodes on the first level of descendent nodes. Each of these nodes represents the assignment of task i, 1 =< i =< n, to be the first in the sequence. Associated with such node there is the completion time t^{ij}, of the task j in the position i, i.e., $t^{1i} = a_i + d_i$. Next we branch from each node on the first level to (n-1) nodes on the second level. Each of these nodes represents the assignment of each of the (n-1) unassigned tasks to be second on the sequence. As before, we associate the corresponding node the completion time of the task $t^{2j} = \max(t^{1i}, a_j) + d_j$. We continue in a similar fashion. The initial node is a dummy node, in the unconstrained case all the node must be present in the level 1 (level 0 is assumed to be the dummy root of the complete tree), in case with precedence constraints in the level 1 we allocate only the tasks that have only external input or no predecessor.

Consider the (n-k+1) new nodes generated at the level k of the tree construction, if the finish time t^{ki} associated with at least one of these nodes exceeds its due date then the subtree rooted at each one of the nodes that are unfeasible may be excluded from further consideration.

The bounding condition applies only when there are no precedence constraints and is intended to find an optimal (minimizing the length of the block) ordering of the sequence. Figure 7 illustrates the application of this criteria.



Tasks

i	1	2	3	4
a_i	4	1	1	0
b_i	7	5	6	4
d_i	2	1	2	2

Figure 7 Example of Scheduling with Earliest Start Time (Nonpreemptible)

In the case with precedence constraints this algorithm does not guarantee an optimal solution, another disadvantage is the time complexity which tends to factorial in the number of tasks. A more detailed explanation, as well, a step by step definition of the algorithm, may be found in [Ref. 20 : p. 514-519].

This algorithm is implemented in this thesis including the precedence constraints. It utilizes the concepts: length of the harmonic building block and the firing interval for each task which are described before in this chapter. The implementation details are explained in Chapter III of this thesis.

4. THE EARLIEST DEADLINE SCHEDULING ALGORITHM

This algorithm also considers the scheduling of n tasks on a single processor. It is a varient of the Earliest Start Scheduling Algorithm, only the earliest deadline should be considered as the criteria instead of the earliest start time. The implementation details are explained step by step in the Chapter III of this thesis.

5. MINIMIZE MAXIMUM TARDINESS WITH EARLY START TIMES SCHEDULING ALGORITHM

This algorithm considers a sequencing problem consisting of n tasks and a single processor. Task i is described by the following parameters :

- the ready time (a_i), the earliest point in time at which processing may begin on i (i.e., an earliest start time).
- the processing time (d_i), the interval over which task i will occupy the processor.
- the due date (b_i), the completion deadline for task i.

The three characteristics a_i, d_i, and b_i are known in advance and no preemption is allowed in the processing of the tasks.

As a result of scheduling, task i will be completed at time C_i and will be tardy if $C_i > d_i$. The tardiness of task(T_i) is defined by $T_i = \max \{0, C_i - d_i\}$. The scheduling objective is to minimize the maximum task tardiness, which is simply $T_{max} = \max_j \{ T_j \}$.

For the static version of the n tasks single processor problem without precedence constraints(all a_i s are equal), T_{max} is minimized by the sequence $b_{[1]} =< b_{[2]} =< \ldots =< b_{[n]}$, that is, by processing the tasks in nondecreasing order of their deadlines.[Ref. 21: p. 172]

In the dynamic version of the problem, the statement above can also be applied if the tasks can be processed in a preemptable fashion, in this case sequencing decisions must be considered both at task completion and at task ready time. Then we have the following :

- At each task completion, the task with minimum b_i among available tasks is selected to begin processing.
- At each ready time, a_i, the deadline of the newly available task is compared to the deadline of the task being processed. If b_i is lower, task i preempts the task being processed otherwise the task i is simply added to the list of available tasks.

The solution to the preemptive case is not difficult to construct because the mechanism is a dispatching procedure. Since all nonpreemptive schedules are contained in the set of all preemptive schedules, the optimal value of T_{max} in the preemptive case is at least a lower bound on the optimal T_{max} for the nonpreemptive schedules. This principle is the basis for the algorithm.

In the nonpreemptive problem, there is a sequence corresponding to each permutation of the integers 1, 2, ..., n. Thus there are at most n! sequences, but several of these sequences do not need to be considered. The number of feasible sequences depends on the data in a given problem, but will be usually less than n!.

A branch and bound algorithm will be used to systematically enumerate all the feasible permutations.

The branching tree is essentially a tree of partial sequences. Each node in the tree at level k corresponds to a partial permutation containing k tasks. Associated with each node is a lower bound on the value of the maximum tardiness which could be achieved in any completion of the corresponding partial sequence (obtained using the preemptive adaptation). The calculation of lower bound allows the algorithm to enumerate many sequences only implicitly. If a complete sequence has been found with a value T^{max} less than or equal to the bound associated with some partial sequence, then it is not necessary to complete the partial sequence in the search for optimum solution.

The branch and bound algorithm maintains a list of nodes ranked in nondecreasing order of their lower bounds. At each stage the node at the top of the list is removed and replaced on the list by several nodes corresponding to augmented partial sequences. These are formed by appending one unscheduled task to the removed partial sequence. The algorithm terminates when the node at the top of the list corresponds to a complete sequence. At this point, the complete sequence attains a value of T_{max} which is less than or equal to the lower bound associated with every partial sequence remaining on the list, and the complete sequence is therefore optimal.

Before the tree search begins, the algorithm uses a heuristic initial phase to obtain a feasible solution to the problem. This initial feasible solution allows the tree search to begin with a complete schedule already on hand, and allows several partial schedules to be discarded in the course of the tree search, simply because their bound exceed the value of the initial solution.

There are four heuristics which can be used:

- Ready time : sequence the tasks in nondecreasing order of their ready time, $a_{i.}$
- Deadline : sequence the tasks in nondecreasing order of their deadlines, $b_{i.}$
- Midpoint : sequence the tasks in nondecreasing order of the midpoints of their ready times and deadlines $(a_i + b_i)/2$. hence use the nondecreasing order of $a_i + b_i$.
- PIO : sequence the tasks in the order of their first appearance in the optimal preemptive schedule, which is constructed by the dynamic version.

The [Ref. 21: pp. 171-176] contains a complete and detailed description of the algorithm as also an analysis of the performance of the algorithm[2]. Considering each heuristic, the global time complexity of this algorithm is $O(n^2)$. As can be visualized, this algorithm does not take into account the possible precedence constraints among the tasks, these precedence constraints must be taken into account during the evaluation of the branch and bound solution of the tree search. The inclusion of the precedence constraints in the evaluation of the heuristics must also be considered. The algorithm can be extended to handle the case where tasks can be started only after some instance of time in the future (this happens when some of the tasks are periodic), the modification necessary is in the definition of task's scheduled start time.

[2]When all tasks are available simultaneously the [Ref. 22: pp.187-199] presents some useful algorithms and an experimental comparison among them, also in [Ref. 23: pp.177-185] we may find some simple and quick algorithms for the same set of conditions.

6. THE RATE-MONOTONIC PRIORITY ASSIGNMENT SCHEDULING ALGORITHM

This algorithm assumes the following premises :

- The requests for all the tasks for which hard deadlines exist are periodic, with period (p_i).
- Deadlines consist of run-ability constraints, that is each task must be completed before the next request for it occurs.
- The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.
- Run-time for each task is constant (d_i) and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

An important concept in determining the rule is that of the critical instant for a task. The deadline of a request for a task is defined to be the time of the next request for the same task. The response time of a request for a certain task is defined to be the time span between the request and the end of the response to that request. A critical instant of a task is defined to be an instant at which a request for that task will have the largest response time. A critical time zone for a task is the time interval between a critical instant and the end of the response to the corresponding request to the task.

Based on the definitions above it is possible to infer that a critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks. One of the values of this result is that a simple direct calculation can determine whether or not a given priority assignment will yield a feasible scheduling algorithm. Specifically, if the requests for all tasks at their critical instants are fulfilled before their respective deadlines, then the scheduling algorithm is feasible. As an example consider two tasks T_1 and T_2 with $p_1 = 2$, $p_2 = 5$, and $d_1 = 1$, $d_2 = 1$. If we let T_1 be the higher priority task then from Figure 8 (a) on page 25 we see that such priority assignment is feasible. Moreover, the value of T_2 can be increased at most to 2 but not further as illustrated in Figure 8 (b). On the other hand, if we let T_2 be the higher priority task, then neither of the values of d_1 and d_2 can be increased beyond 1 as illustrated in Figure 8 (c).

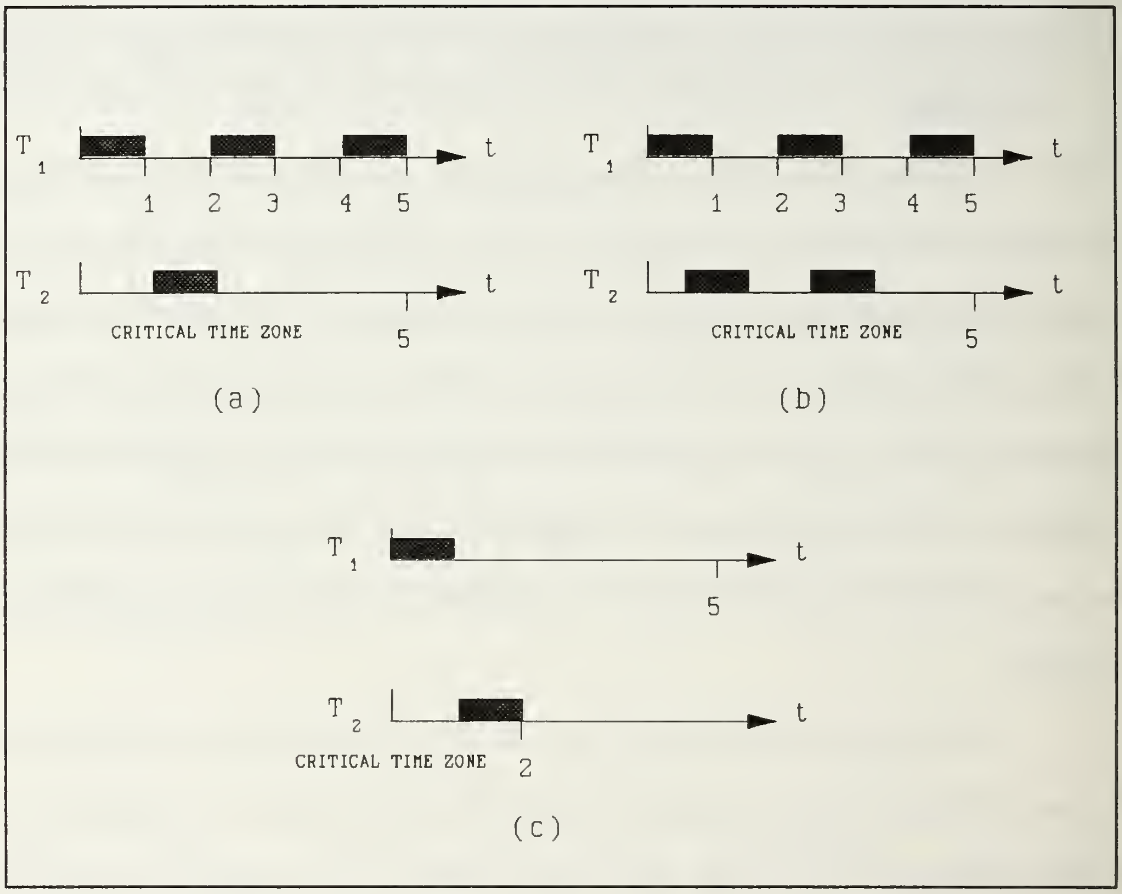


Figure 8 Schedule for Two Tasks

The analysis of the example above suggests a priority assignment. Let p_1 and p_2 be the request periods of the tasks, with $p_1 < p_2$. If we let T_1 be the higher priority task then, according to the definition of critical instant, the following inequality must be hold $\lfloor p_2/p_1 \rfloor d_1 + d_2 =< p_2$[3].

If we let T_2 be the higher priority task, then, the following inequality must be satisfied $d_1 + d_2 =< p_1$. In other words, whenever the $p_1 < p_2$ and d_1, d_2 are such that the task schedule is feasible with T_2 at higher priority than T_1, it is also feasible with T_1 at higher priority than T_2, but the opposite is not true. Thus we should assign a higher priority to T_1 and lower priority to T_2. Hence, more generally, it seems that a reasonable rule of priority assignment is to assign priorities to tasks according to request rates, independent of their run-times. Specifically, tasks with higher request rates will have higher priorities. Such an assignment of priorities is known as the Rate-Monotonic Priority Assignment. Such priority assignment is optimum in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by the rate-monotonic priority assignment.

A formal development and analysis of this algorithm, as well the theoretical development of maximum achievable processor utilization of this type of algorithm is available in Liv [Ref. 25: pp. 46-61].

Some algorithms for scheduling periodic tasks to minimize average error utilizes the rate-monotonic priority assignment algorithm in order to solve the scheduling of the mandatory part of all the tasks, a complete description of these algorithms may be found in [Ref. 26: pp. 142-150].

---

[3]This condition is necessary but not sufficient to guarantee the feasibility of the priority assignment. The symbol $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x.

C. SUMMARY

This survey presented some of the previous single processor static scheduling algorithms for hard real-time systems. Many of the algorithms discussed do not address the problem of how to schedule tasks that have precedence constraints. When it was necessary to obey an earliest ready time, usually an algorithm based in a tree branch and bound was used. The concept of a cost function to evaluate the schedule was shown in the minimize maximum tardiness with early start times scheduling algorithm. When precedence constraints were considered in the algorithms, the solution adopted was to use some kind of graphical representation (directed graphs), and the notion of a base timeframe was used (harmonic block). None of the algorithms presented gives an optimal solution to the problem of scheduling hard real-time system with precedence constraints. A general survey of Static Scheduling Algorithms can be found in Cervantes [Ref. 33].

The approach that will be followed in this thesis is to develop the ideas exposed in the harmonic block with precedence constraints scheduling algorithm (in order to define a timeframe), and implement the three of the algorithms presented in this chapter.

III. IMPLEMENTATION OF STATIC SCHEDULERS

When we looked at the history in developing the implementation of the Static Schedulers we see some variations in basic data structures used. The first guidelines about the Static Schedulers' current implementation were outlined in O'Hern [Ref. 14]. O'Hern introduced the "Graph Type Model" developed by Mok and Sutanthavibul [Ref. 28] as a basic unit. Johnson [Ref. 13] wrote the first pseudo code with some deviations from O'Hern. Then Marlowe [Ref. 6] did a part of the first implementation of the basic design. In her implementation, the tree structure was used as a basic unit. In this chapter, the implementation of the basic design which is declared as "The Harmonic Block with Precedence Constraints Scheduling Algorithm" has been completed with some deviations from Marlowe's [Ref. 6]. Besides, two other algorithms, The Earliest Start Scheduling Algorithm and The Earliest Deadline Scheduling Algorithm are also implemented. In the implementation of the Static Scheduling algorithms in this thesis, Ada® Language has been used as a basic language. The Appendix D of this thesis has the dependency information of the programs implemented, and Appendix E has the Ada source code of all the programs and data structures utilized.

A. ASSUMPTIONS

First, this design assumes that the PSDL Prototype is syntactically correct. This implies that each line begins with a PSDL keyword or reserved word. Second, the designer structured the PSDL prototype program using a top-down design. This implies that the program begins with the highest level and then decomposes all composite operators, with the last(or lowest) level being the Ada® implementation modules. The implementation design in this thesis addresses a single processor environment only. All operators are nonpreemptable, and except non-time critical operators, all critical operators should have a Maximum Execution time(MET). If the operators are sporadic, they have an MET, Maximum Response Time(MRT) and a Minimum Calling Period(MCP). It is also assumed that all timing constraints are non-negative integer values. The system may include state machines, and external inputs and outputs. It can handle the acyclic digraphs as linear digraphs. The data coming in from any External input is assumed ready at execution time. The implemented algorithms use the precedence relationships between the operators. The Static Schedulers implemented here only accept the critical timing information extracted from the output file of "PSDL_Reader". Normally this Text File has the timing and link information of the atomic operators only.

B. DATA STRUCTURES UTILIZED

The major data structure used in the current implementation of static schedulers utilizes Graph Type Model. This model is defined in [Ref. 28] and explained in [Ref. 14]. For this model, a Graph Type is created by using a generic type Graph Package. Five data type abstractions are used in current implementations. They are as follows:

- OPERATOR
- LINK_DATA
- THE_GRAPH

- SCHEDULE_INPUTS
- OP_INFO

OPERATOR contains all the critical timing information of each operator extracted from the "atomic_info" file. LINK_DATA contains the link information among the operators and is utilized in THE_GRAPH. THE_GRAPH is the basic unit of the static schedulers in this thesis. SCHEDULE_INPUTS contains the scheduling information of all operators and is used to create the final output.

Data types and their corresponding data structures are as follows :

Abstract Data Types	Data Structures
OPERATOR	Linked List
LINK_DATA	Linked List
THE_GRAPH	Graph
SCHEDULE_INPUTS	Linked List
OP_INFO	Linked List

OPERATOR, SCHEDULE_INPUTS, and OP_INFO, as global data types, are encapsulated in an Ada® package called FILES which allows the other packages to use them directly. The LINK_DATA and THE_GRAPH are utilized in an Ada® generic package called GRAPHS which is used to create the Graph Structure for the Static Scheduler in FILES. So the complete structure is created in package FILES. Files were only used for the storage of information that would be used outside of the Static Scheduler by the Execution Support System.

1. LINKED LISTS

A single operator is implemented with type OPERATOR as a record with six fields as originally designed [Ref. 6]. These fields are shown in Table 1. Although it is not necessary to fill all the fields in the record for all the operators, these fields are required as a whole considering the different type of operators(e.g. periodic and sporadic). Section E of this chapter explains the required fields in details. It is the basic unit to store the atomic operator information within a Linked List in Graph Structure. It is also utilized to construct a precedence list in the implementation of the first algorithm.

Table 1 Record Fields for OPERATOR

FIELDS	CONTENTS
THE_OPERATOR_ID	The name of the operator
THE_MET	maximum execution time for the operator
THE_MRT	maximum response time for the operator
THE_MCP	minimum calling period for the operator
THE_PERIOD	the operator's period
THE_WITHIN	the time within which the operator must finish

A single instance of the type LINK_DATA was implemented as a record with four fields. These four fields are shown in Table 2. This is the basic unit of the link information, which is implemented as a Linked List in the graph. The link information of the graph is available in the input text file and the Linked List is constructed. A defined order is not required for the Linked List which stores the link information in Graph. Figure 9 shows the relationship between the Graphical and Data Structure representation in a link statement.

The third abstract data type used in the Static Scheduler is SCHEDULE_INPUTS. It is a record which consists of five fields. These fields are shown in TABLE 3. It has the final scheduling information about each operator and it is utilized to create the static schedule output.

Table 2 Record Fields for LINK_DATA

FIELDS	CONTENTS
THE_DATA_STREAM	The name of the link
THE_FIRST_OP_ID	Start of the link
THE_LINK_MET	Maximum execution time for data transfer
THE_SECOND_OP_ID	End of the link

Table 3 Record Fields for SCHEDULE_INPUTS

FIELDS	CONTENTS
THE_OPERATOR	The name of the operator
THE_START	Start time for the execution
THE_STOP	Stop time for the execution
THE_LOWER	Lower bound for the firing interval
THE_UPPER	Upper bound for the firing interval

OP_INFO is the last abstract data type which is used in the "Earliest Start Scheduling Algorithm" and "Earliest Deadline Scheduling Algorithm". The fields are shown in Table 4. Detailed Linked List representation will be given in Section F.

The Linked Lists used in this implementation is constructed by using an Ada generic package called SEQUENCES, so that any data type could be stored in the nodes of the list. The SCHEDULE_INPUTS_LIST, V_LISTS, E_LISTS, and OP_INFO_LIST in the generic Graph package are constructed by using SEQUENCES. The required functions and procedures are encapsulated in SEQUENCES generic package which enables the user to operate on the List without knowledge of its internal structure.

Table 4 Record Fields for OP_INFO

FIELDS	CONTENTS
NODE	The operator information
SUCCESSORS	Successors of the operator defined in the NODE
PREDECESSORS	Predecessors of the operator defined in the NODE

These operations include, but are not limited to, the following :

- EQUAL -- determine if the two lists are equal to each other
- EMPTY -- create an empty list
- NON_EMPTY -- determine if the list is empty
- SUBSEQUENCE -- determine if a list is a subsequence of the original list
- MEMBER -- determine if the operator is in the list
- ADD -- add the operator into the list
- REMOVE -- remove the operator from the list
- LIST_REVERSE -- reverse the order of the original list
- DUPLICATE -- duplicate the original list

- LOOK4 -- determine if the operator is in the list
- NEXT -- point to the next operator in the list
- VALUE -- return the operator record values.

The complete specification and implementation of this Linked List can be found in Appendix E.

2. GRAPH

The Graph type represents the Graph Type Model and has the complete information about the Graph, including operators and links information. Figure 8 shows how the graph type is implemented. It only presents the information required according to the operators being either periodic or sporadic. It is a record which consists only two fields, VERTICES and LINKS. They are shown in TABLE 5. VERTICES is a pointer for the V_LISTS which is a linked list to store the operators information and uses the OPERATOR type as a basic unit, and the LINKS is a pointer for the E_LISTS which stores the link information and uses the LINK_DATA type.

Table 5 Record Fields for DIGRAPH

FIELDS	CONTENTS
VERTICES	Operator list of the graph
LINKS	Link list of the graph

The graph model is constructed by using an Ada Generic Package called GRAPHS, so that any data type could be stored in the nodes of the graph. In the case of the Static Scheduler, the nodes are of the type OPERATOR. The required functions and procedures were encapsulated in GRAPHS generic package enabling the user to operate on the graph without knowledge of its internal structure. These operations include, but are not limited to, the following :

- EQUAL_GRAPHS -- determine if the two graphs are equal to each other
- EMPTY -- creates an empty graph
- IS_NODE -- determine if the operator is in the graph
- IS_LINK -- determine if a link is in the graph
- ADD -- add a link into the graph
- ADD -- add an operator into the graph
- REMOVE -- remove a link from the graph
- REMOVE -- remove an operator from the graph
- SCAN_NODES -- search the graph for a given operator
- SCAN_PARENTS -- find the parents of a given operator in the graph
- SCAN_CHILDREN -- find the children of a given operator in the graph
- DUPLICATE -- duplicate the given graph
- T_SORT -- sort the operators of the graph in a topological order.

Operations on the graph are easy to use. The use will be explained in details later in this Chapter. A complete listing of the specification and implementation of the Graph can be found in Appendix E.

3. VARIABLE LENGTH STRINGS

The Ada language has a predefined "string" type, but this couldn't be used as the base type for the operator and data stream fields within the OPERATOR, LINK_DATA, and SCHEDULE_INPUTS types, because the string must have a predefined fixed length. Since these fields are necessarily of a variable length, to accommodate the Ada identifiers that would be assigned to them, a variable length string abstract data type was necessary. A generic variable length string package from a public domain library was chosen for the implementation. It has functions to convert a standard Ada string to a variable length string, functions for comparison, and procedures for input and output. These were the main functions necessary for the static scheduler, though there are many others in the package.[Ref. 6] Utilization of the package is very simple, and a complete listing of the specification and implementation for the variable length strings abstract data type can be found in Appendix E.

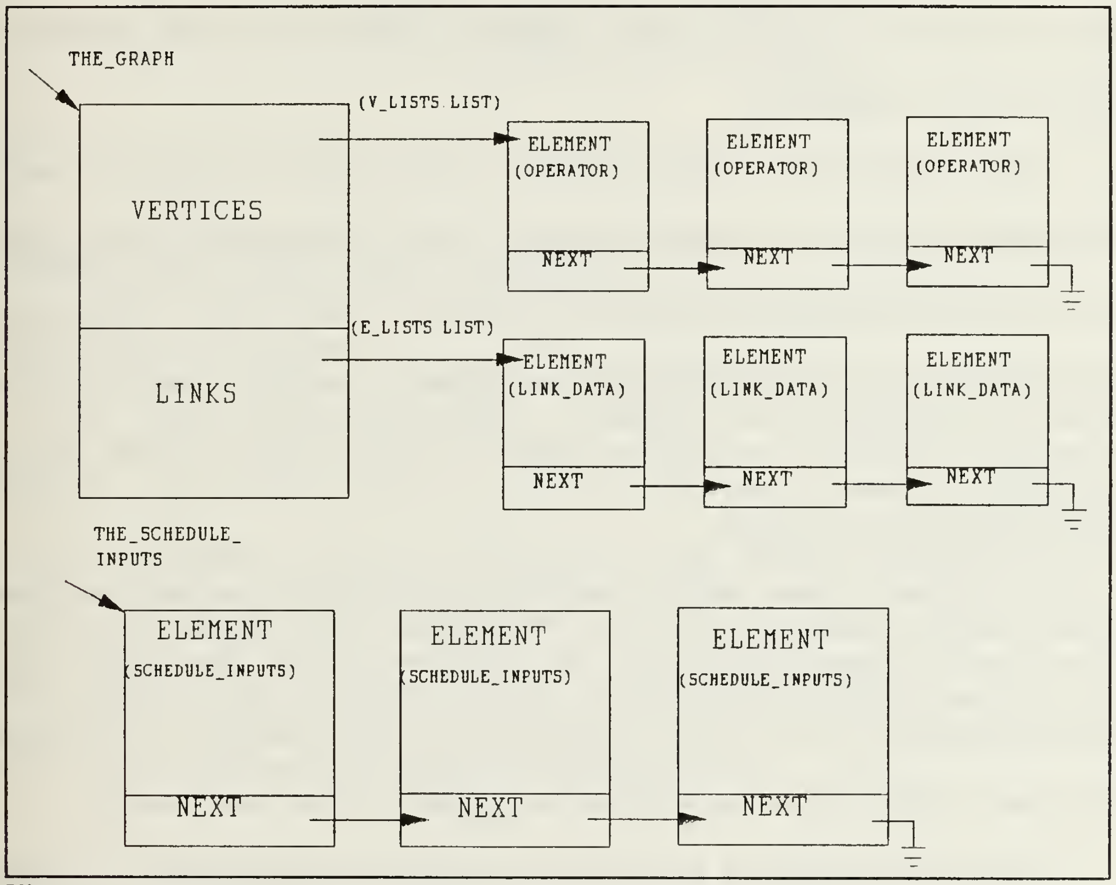


Figure 9 Graphical Representation of the system and the data types used

C. ARCHITECTURAL DESIGN FOR STATIC SCHEDULERS

The general DFD for the Static Schedulers is shown in Figure 2. Although there are strong similarities with the original static scheduler for CAPS, the architectural design is slightly modified to allow the system to run more than one algorithm and simplify the decomposition process. The "PSDL_Reader" in described in Chapter II is called "Preprocessor" in White [Ref. 30] and in this thesis. The "Preprocessor" and "Decomposer" were not implemented in this thesis. An example graph with its "PSDL_Reader" output file and the file which would be the output of the "Decomposer" were given in Appendix C. Except the "Topological_Sorter" module which is used only by the first static scheduling algorithm, the other modules are shared by all the algorithms.

In this design the first module, known as "File_Processor", reads the input file "atomic.info" which has the timing constraints and link information of the operators, and extracts the information in this file to construct the Graph Structure. The operators which have no critical timing information are separated to another output file, referred as "non_crits". This file is used by the Dynamic Scheduler which schedules non-time critical operators for execution.

The "Harmonic_Block_Builder" module first calculates the periodic equivalents of the sporadic operators which have no predefined periods. Then checks, if an Harmonic Block can be found for a single processor. If yes, it calculates The Harmonic Block Length, which is used to schedule the operators in their time intervals.

The module "Topological_Sorter" takes the Graph Structure as an input and builds a precedence relationship, that specifies which operators must complete execution before

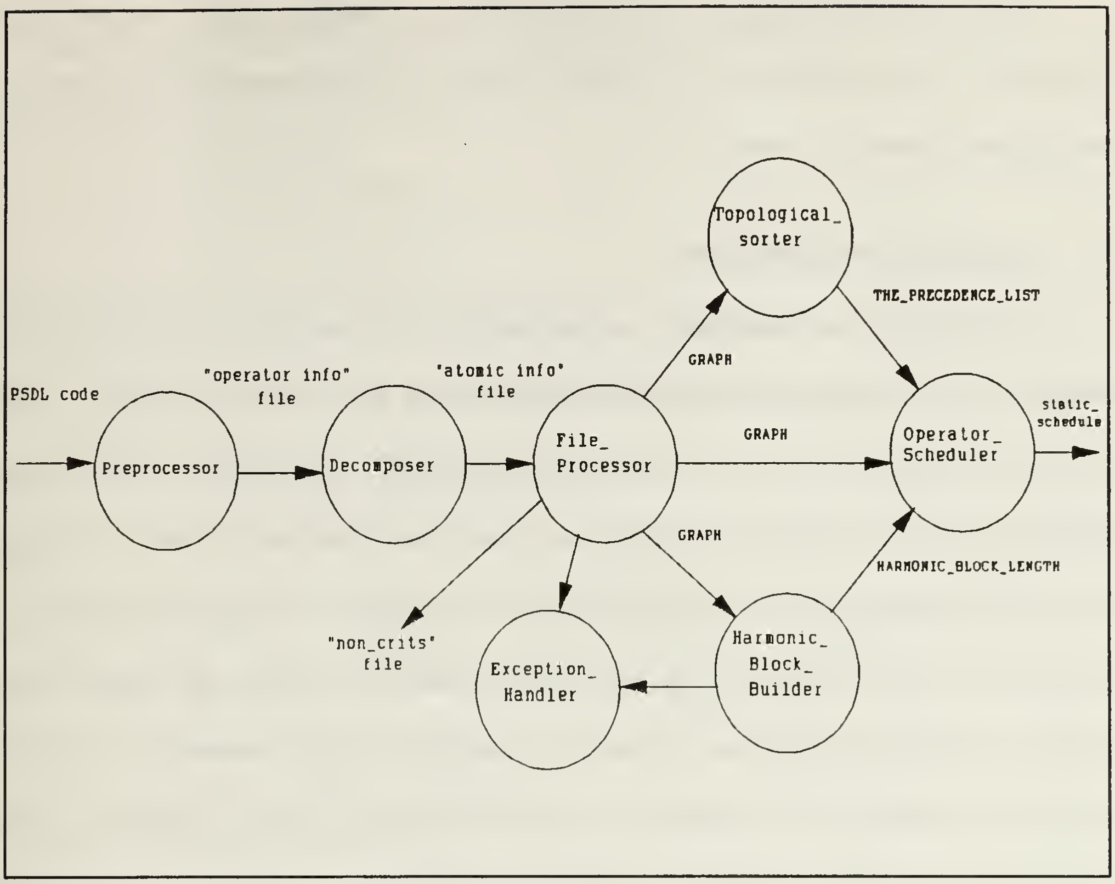


Figure 10 New DFD for Static schedulers

other operators can execute.

The module "Operator_Scheduler" combines the Precedence List and the Harmonic Block Length for the for the first algorithm to produce a final Static Schedule, if possible. Since the Earliest Start and Earliest Deadline Scheduling Algorithms do not need THE_PRECEDENCE_LIST, they use only the graph structure and the Harmonic Block Length. To keep the design DFD as simple as it is, all the static scheduling algorithms are included in this module.

The "Exception_Handler" is the last module and handles all the exceptions which are critical for the execution of the Static Scheduler. It terminates the program to let the designer correct the errors.

D. EXCEPTION HANDLING

In this thesis, the schedulers are designed in order to build a static schedule by using the atomic operator information extracted from the "atomic.info" input text file, unless the conditions are found which would make the construction of the schedule infeasible. If none of these conditions are found, the schedulers construct a schedule for all the operators that were known for the system. During the operation, an exception is raised in two conditions. One of them is to notify the designer that a schedule is infeasible with the information provided, if any condition is found that makes the construction of a schedule inpossible. In this case the scheduler terminates the execution. The other one is to notify that although a schedule may be possible, there is no guarantee that it will execute within the required timing constraints. In both cases. In this case, the scheduler tries to find a feasible solution without terminating the execution.

As we know, Ada® includes several predefined exception conditions, but it also permits us to declare user-defined exceptions. Although an exception is technically not an object, user-defined conditions may be declared anywhere an object declaration is appropriate (except as a subprogram parameter).[Ref. 29]

Three different types of exception handling will be noticed throughout the implementation, which are shown in Table 6. Number 1 through 3 are the examples

Table 6 Exceptions used in Static Schedulers

1.	MISSED_DEADLINE
2.	OVERTIME
3.	MISSED_OPERATOR
4.	NO_BASE_BLOCK
5.	CRIT_OP_LACKS_MET
6.	MET_NOT_LESS_THAN_MRT
7	MCP_NOT_LESS_THAN_MRT
8.	MCP_LESS_THAN_MET
9.	SPORADIC_OP_LACKS_MCP
10	SPORADIC_OP_LACKS_MRT
11.	MET_NOT_LESS_THAN_PERIOD
12.	MET_IS_GREATER_THAN_FINISH_WITHIN
13.	PERIOD_LESS_THAN_FINISH_WITHIN
14	BAD_TOTAL_TIME
15.	FAIL_HALF_PERIOD
16.	RATIO_TOO_BIG

of the first type and used to notify the designer that there is no feasible schedule exists which meets the requirements of the system in the running scheduling algorithm. This type of exception is handled inside the driver program to allow more than one static scheduler to run. The second type of exception handling is used to raise exception in the local program unit, but passes exception handling to the driver program. In this case, when the Static Scheduler discovers an exception, the following occur. A variable, named Exception_Operator, is set by the Static Scheduler and a procedure call to the Static Scheduler Exception Handler is made to transfer control to the Exception Handler. This allows the Exception Handler to handle the exception and gives the designer the name of the operator that caused the exception. This is done in the Static Schedulers by having a global variable named "Exception_Operator" set by the local programs before any of this type of exception condition is discovered. This shows that a schedule is infeasible with the information set provided, which means the scheduler will end the execution without producing a schedule, and thus lets the designer make the corrections. Exceptions 4 through 13 indicate that either required constraints are missing or they are logically inconsistent. These are the examples of the second type. The third type also has the concept of "Exception_Operator" as the second type, it is handled inside the packages and its only function is to change a global variable, "Exception_Operator", and print a descriptive message. Exceptions 14 through 16 indicate that, a feasible schedule may be possible, but there is no guarantee that it will execute within the required timing constraints These are the examples of the third type.

E. PACKAGE PRESENTATIONS OF "THE HARMONIC BLOCK WITH PRECEDENCE CONSTRAINTS SCHEDULING ALGORITHM"

This Static Scheduler, as implemented in this thesis, contains six package programming units. Four packages represent primary functional groupings, with two additional packages EXCEPTION_HANDLER and FILES The EXCEPTION_HANDLER package has the exception-handling procedures used by all the other packages, which are called by the driver program, and the FILES contains global data type declarations. The packages utilized in this algorithm are described below:

1. "FILES" Package

The variable length string, discussed earlier in this chapter, is in the package because it is an essential data structure for the implementation. It enables the operator names and the data streams of variable length in the implementation, up to a maximum of 80 characters. The number of characters was chosen arbitrarily and can be changed, however, it seems that an Ada identifier of more than 80 characters wouldn't be necessary.

All the values used for the critical timing information within the data types are natural numbers to correspond with PSDL, which makes comparison of values within these fields simpler; which in turn would be important when the algorithms were utilized in CAPS.

All the packages are instantiated for each of the data types given. This includes the DIGRAPH for the graph structure, and linked list for the SCHEDULE_INPUTS. This encapsulation of the major data structures allows the rest of the packages to proceed.

2. "FILE_PROCESSOR" package

This module has two procedures in it, SEPARATE_DATA and VALIDATE_DATA. All the identified exceptions in the procedure VALIDATE_DATA in the FILE_PROCESSOR package include :

1. CRIT_OP_LACKS_MET
2. MET_NOT_LESS_THAN_MRT
3. MCP_NOT_LESS_THAN_MRT
4. MCP_LESS_THAN_MET

5. SPORADIC_OP_LACKS_MCP
6. SPORADIC_OP_LACKS_MRT
7. MET_NOT_LESS_THAN_PERIOD
8. MET_IS_GREATER_THAN_FINISH_WITHIN
9. PERIOD_LESS_THAN_FINISH_WITHIN

The non-time critical operators are separated in SEPARATE_DATA and put into the "non_crits" file for future use in Dynamic Scheduler. While the non-time critical operators are separated, all its dependent link information is also checked and extracted without putting them into the graph structure. It is assumed that time critical operators always have an MET and non-time critical operators never have any time constraints. All the periodic and sporadic operators are extracted from the input text file in SEPARATE_DATA and a Graph structure is constructed. This procedure also extracts the EXTERNAL input and output link information in that file.

The example shown in Appendix. B for the Fig. 14 is an Acyclic type of graph. In the graph, OP_3 is a sporadic operator and OP_5 is non-time critical. It has EXTERNAL input and output data streams with the two data streams from OP_1 to OP_2. The current implementation of the static scheduler will extract the non-time critical operator OP_5 from the graph by using SEPARATE_DATA procedure in FILE_PROCESSOR package and put it into the "non_crits" file. The EXTERNAL input-output data streams are assumed ready whenever needed, the graph doesn't have this information either. Fig. 11(b) shows the latest form of the graph structure. The links which are related with this non-time critical operator are excluded from the graph later in the same procedure. OP_3 is converted into its periodic equivalent with the CALC_PERIODIC_EQUIVALENTS procedure in HARMONIC_BLOCK_BUILDER package.

The procedure VALIDATE_DATA is one of the most important procedures within the static scheduler. Static Scheduler performs some basic validity checks on the timing constraints contained in the "atomic.info" file, which is accomplished after the Graph structure is built. The first check CRIT_OP_LACKS_MET verifies that all critical operators have an MET. Checks 2 through 6 are valid for Sporadic Operators; if the Sporadic Operator doesn't have an MCP, the exception SPORADIC_OP_LACKS_MCP is raised, or else MCP_LESS_THAN_PERIOD ensures that MCP is less than MET. The SPORADIC_OP_LACKS_MRT ensures that MRT has a value and MET_NOT_LESS_THAN_MRT ensures that MRT is greater than the MET for the Sporadic Operators. The MCP_NOT_LESS_THAN_MRT guarantees that an operator can fire at least once before a response expected. The significance of these validity checks will become apparent in the section for "HARMONIC_BLOCK_BUILDER" package. Checks 7 through 9 are for the periodic operators; MET_NOT_LESS_THAN_PERIOD ensures that the PERIOD is greater than MET, MET_IS_GREATER_THAN_FINISH_WITHIN ensures that FINISH_WITHIN is greater than MET, and PERIOD_LESS_THAN_FINISH_WITHIN is included for the correct execution of the algorithms. In all nine cases, if any one of these checks fails, an exception is raised and an appropriate error message is submitted to the user.

3. "TOPOLOGICAL_SORTER" package

The TOPOLOGICAL_SORTER package contains only one procedure which utilizes T_SORT in the generic GRAPH package. It is a simple algorithm that essentially finds the operator, which must precede all others in a set, concatenates that operator to a sequence of operators, which is called PRECEDENCE_LIST and then deletes this operator and all its incoming and outgoing edges from the graph. This cycle is repeated until all operators have been deleted from the graph. The final sequence in PRECEDENCE_LIST should contain all operator names, in order, by precedence. Fig. 14(a) shows a PSDL graph implementation with its EXTERNAL input and outputs, but this graph is represented as seen in Fig. 14(b) in the graph structure implemented in this thesis, the assumption of incoming data from EXTERNAL sources are ready at start allows us to do this. Since all the links are deleted after the operator was added into the PRECEDENCE_LIST, there wouldn't be any duplicates of the same operator in this list.

4. "HARMONIC_BLOCK_BUILDER" package

The same graph structure is also the input for this package. A time frame in this thesis is a set of periodic operators where the periods off all its component operators are exact multiples of a calculated base period [Ref. 15: p. 7]. This package is implemented as described in Chapter II, Section B, with the exception of sorting of the operators in ascending order, based on the period values after all the operators are in periodic form. Instead, the minimum period is found for calculation of GCD because only the smallest period was required for finding GCD, and this was simpler to implement than sorting the list.

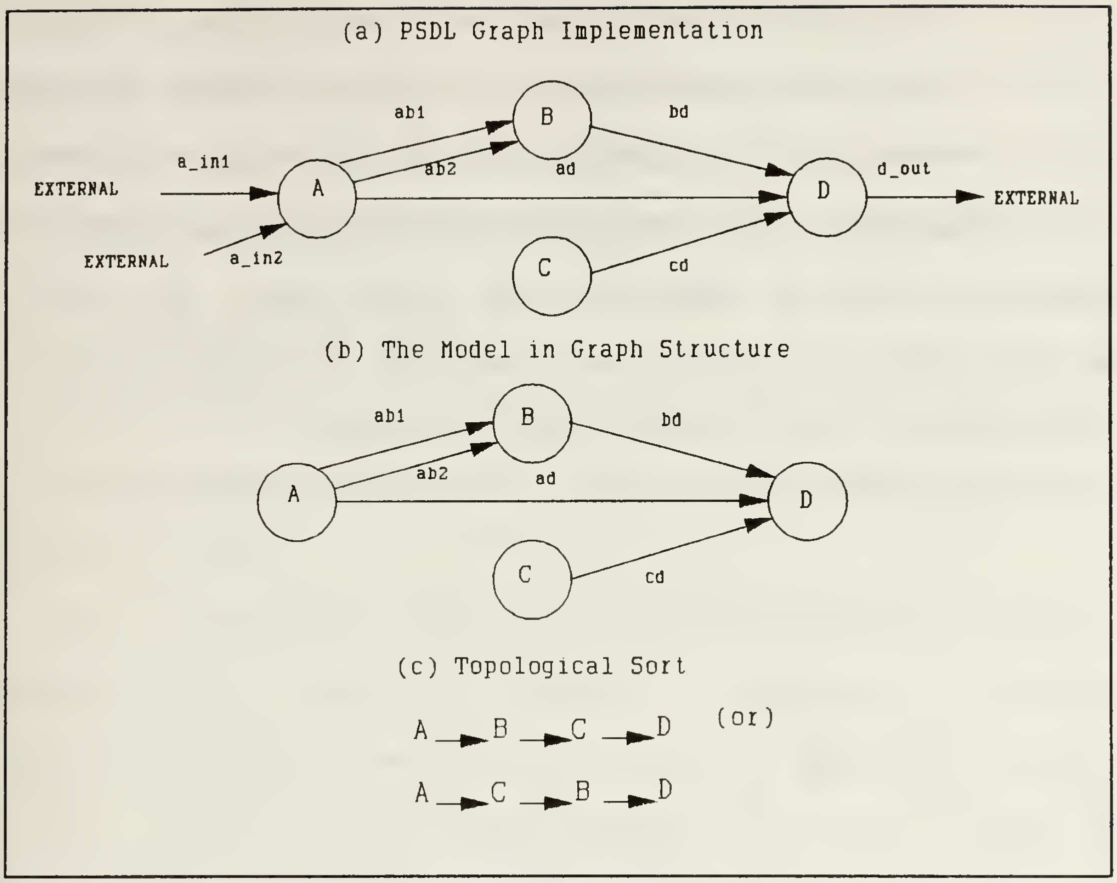


Figure 11 PSDL Graph and its representation in implemented Graph Structure

The procedure CALC_PERIODIC_EQUIVALENTS was used to determine the equivalent periods for sporadic operators. And FIND_BASE_BLOCK was used to find a base block which verifies that an Harmonic Block Length can be determined for the designed system. The two algorithms that can be used to determine the GCD which is described in Janson [Ref. 13: p. 38]. Within this thesis the second algorithm is used since the implementation was more straightforward, and, for a single-processor environment, the second pass verifies that all periods were assigned correctly to the first sequence if the alternate sequence equals the null set [Ref. 13: p. 38]. The last procedure is the FIND_BLOCK_LENGTH which uses an algorithm to calculate the length of time for the Harmonic Block known as The Least Common Multiple(LCM) of all the operators'period contained in the block. Figure 12 describes the algorithm which is explained in detail in Janson[Ref. 13: p. 39]. Two exceptions are reasonable to have in this package. One of them is NO_BASE_BLOCK which means that it is not possible to find a length for the time frame. The other is MET_NOT_LESS_THAN_PERIOD which verifies that the calculated period of the sporadic operator is greater than MET of the same operator.

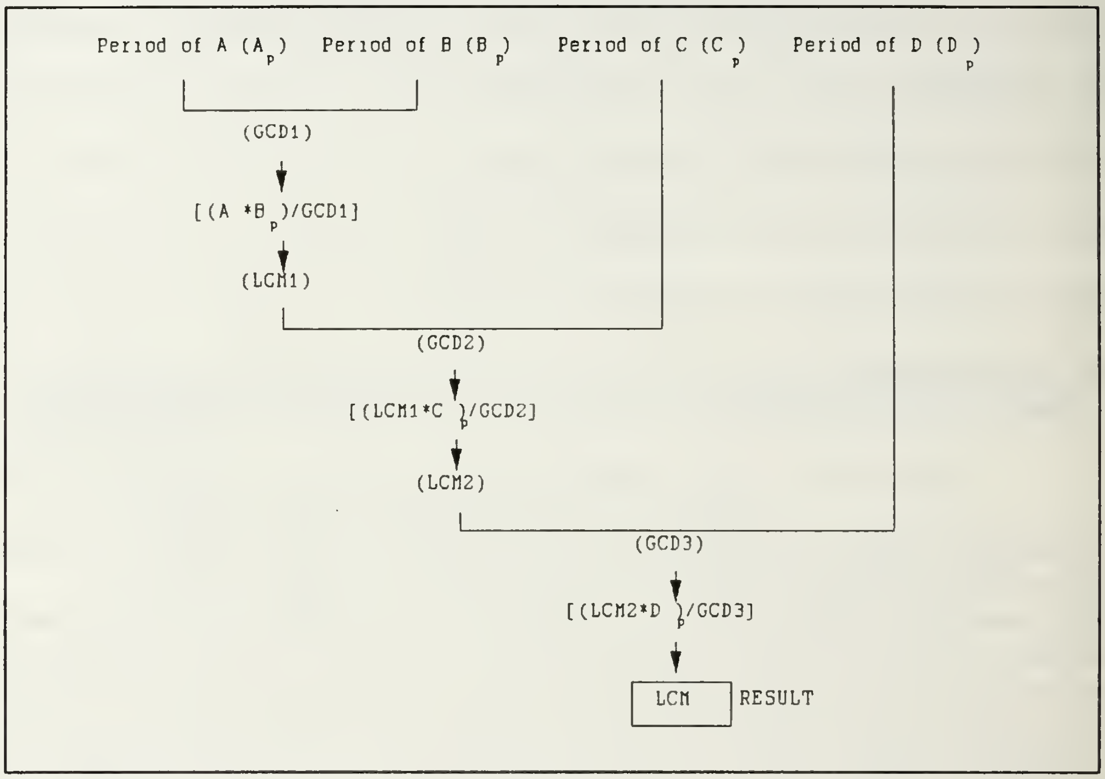


Figure 12 Finding a time interval for the system

5. "OPERATOR_SCHEDULER" package

The PRECEDENCE_LIST and HARMONIC_BLOCK_LENGTH were used as input in the OPERATOR_SCHEDULER for this scheduling algorithm. Procedure

TEST_DATA tests the operators if they follow three basic rules which verifies that a feasible static schedule always exist. These basic rules include:

- The MET of the operator should be less than half of its period
- The total MET/PERIOD ratio sum of operators should be less than 0.5
- Tha total execution time of the operators should not exceed the HARMONIC_BLOCK_LENGTH.

Detailed information can be found in Mok [Ref. 30]. If some of these tests are not satisfied, the static schedulers will try to find a feasible schedule, but there is no guarantee to have one.

Part of the OPERATOR_SCHEDULER which belongs to the first algorithm is implemented in two steps as mentioned in Chapter II; the procedure SCHEDULE_INITIAL_SET performs the first step process, and allocates an execution time with a firing interval for each operator to use in the next step. The SCHEDULE_REST_OF_BLOCK performs the second step and completes the rest of the process. The procedure CREATE_INTERVAL is used by the SCHEDULE_INITIAL_SET in the first step and by the SCHEDULE_REST_OF_BLOCK for the next firing intervals. Appendix A shows the static schedule for the linear graph in Fig. 13 at the end of the process. The operators are scheduled in the order {read_numbers, sort_numbers, write_numbers} during the first iteration of this process. Since all the operators have a period of 20 with a harmonic block length 20, they are scheduled only once in the block. Since all the firing intervals are greater than the harmonic block length in this example, we do not need a second process. Before an operator is allocated a time slot, this process verifies

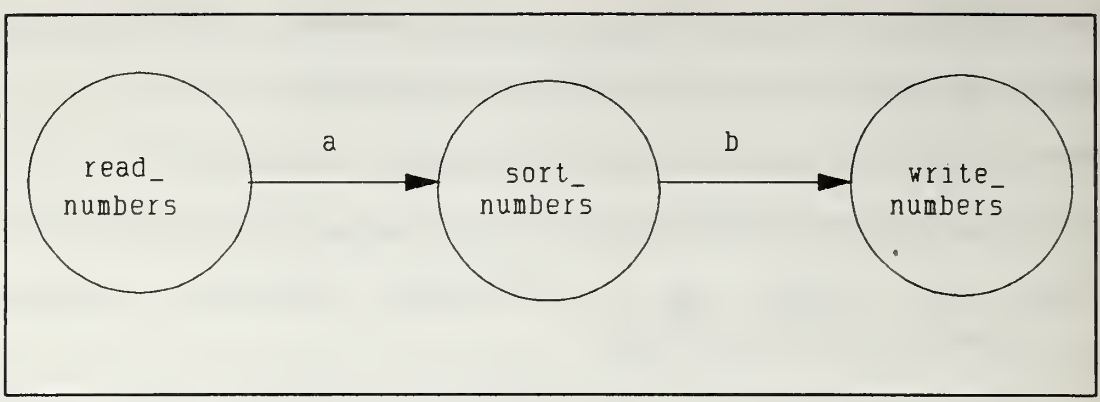


Figure 13 Graph Model for Example 1

for all the operators that:

- (current_time + MET) <= harmonic block length

In the example shown in Appendix B, for Fig. 14 ,we have the second process as the continuation of the first process. In this example, since the OP_2 has a FINISH_WITHIN constraint, this is considered in calculating the firing interval of OP_2. This means that for the upper limit of the intervals of OP_2 the FINISH_WITHIN is used instead of PERIOD.

F. IMPLEMENTATION OF "THE EARLIEST START SCHEDULING ALGORITHM"

The nonpreemptable version of this algorithm is implemented in this thesis, and precedence constraints are included.

This algorithm utilizes all the packages that the previous algorithm does with the exception of TOPOLOGICAL_SORTER. Although this algorithm doesn't use that package, it considers the precedence relationships among the operators, with the way it is implemented in this thesis.

First, the Graph Structure is constructed as being described in previous algorithm, and all the tests in FILE_PROCESSOR package are applied. When the Graphical representation of the system is approved, the Harmonic Block Length is calculated. Then the algorithm starts to deviate from the first algorithm. The rest of this section describes in details, how the algorithm works with the procedures used in the OPERATOR_SCHEDULER.

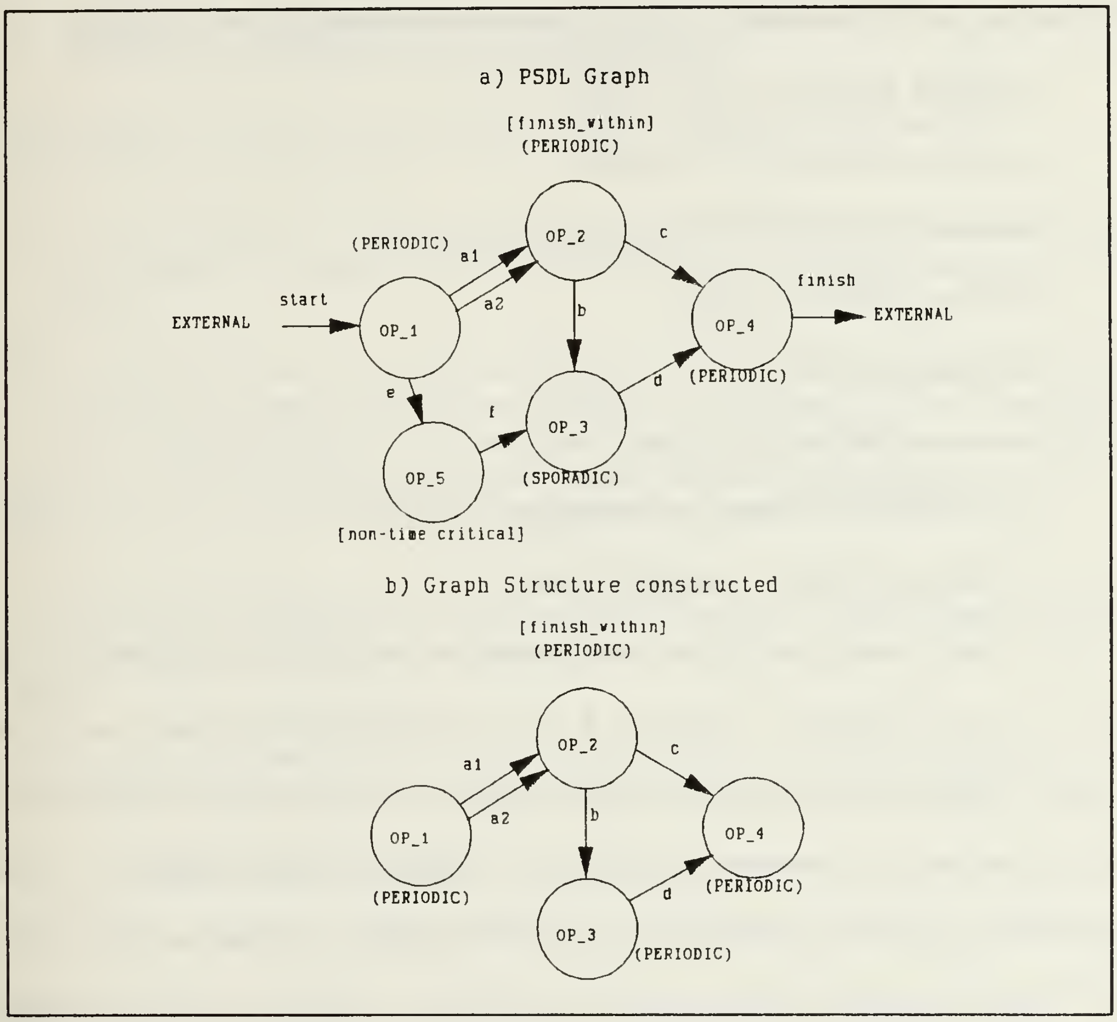


Figure 14 Graph structure for Example 2

1. "OPERATOR_SCHEDULER" package

This is the same package used for the first algorithm. It includes the the procedure for the Earliest Start Time Scheduling Algorithm which is called SCHEDULE_WITH_EARLIEST_START. The final output list(AGENDA) of this procedure is used by the procedure CREATE_STATIC_SCHEDULE for the final output. There are some other functions and procedures that the SCHEDULE_WITH_EARLIEST_START procedure uses. They are as follows:

1. procedure BUILD_OP_INFO_LIST
2. procedure PROCESS_EST_NODE
3. function FIND_OPERATOR
4. function CHECK_AGENDA
5. procedure EST_INSERT
6. function OPERATOR_IN_LIST
7. procedure EST_INSERT_SUCCESSORS_OF_OPT
8. procedure PROCESS_EST_AGENDA

Two examples are shown in Appendix A and Appendix B for Fig. 13 and Fig. 14. In the example in Appendix B, the total MET/PERIOD ratio sum of the operators is greater than 0.5. This message is printed on the screen, but since this is not a fatal constraint, the algorithm proceeds to run for a feasible schedule. As soon as the Time Interval is determined, the OP_INFO_LIST as shown in Fig. 15(a) is constructed in procedure BUILD_OP_INFO_LIST. The AGENDA list includes the final operators list with their start and stop times which are used by CREATE_STATIC_SCHEDULE for the final static schedule, shown in Fig. 15(b), and

MAY_BE_AVAILABLE list includes the available operators with their EST's for the scheduling, shown in Fig. 15(c). The processes of this algorithm are explained in the following steps:

1. Find the operators which has no predecessors and put them all into the MAY_BE_AVAILABLE list. Since all these operators have the same Earliest Start Time(EST), the order of the operators is not important in here. The EST for all of these end nodes is zero. Since EST is the same, we can pick any one of them according to which one is first in the list.

2. Select the first operator and put it into AGENDA list with a calculated start time(THE_START) and stop time(THE_STOP).

3. Define a new EST for the selected operator and put it back into the MAY_BE_AVAILABLE list.

4. Assign THE_STOP of the selected operator to its successors as their EST's and insert them into the MAY_BE_AVAILABLE list in an order according to their EST's.

5. Get the first operator with the smallest EST in MAY_BE_AVAILABLE list and look if all its predecessors are in AGENDA. If the answer is no, then get the next operator and check the predecessors again. Repeat the process until the answer is yes. Then assign a new EST for the selected operator and put it back in the MAY_BE_AVAILABLE_LIST in an order according to its EST.

6. If any successor of the selected operator is not ALREADY in the MAY_BE_AVAILABLE list, assign THE_STOP of the selected operator to that successor as its EST and insert into the MAY_BE_AVAILABLE list in its order.

7. Repeat the process 5 and 6 above until the EST of the selected operator in MAY_BE_AVAILABLE list is greater or equal to the time interval(HBL).

During the implementation of this algorithm, the abstract data types are tried to be utilized instead of creating new data types. This is preferred to avoid the complexity of the programs and reduce the time spent for creating the new data structures. Besides, this was very practical for the comparisons among the operators. As a result of this, the SCHEDULE_INPUTS abstract data type is used for the operators in AGENDA and MAY_BE_AVAILABLE list. For the EST information of

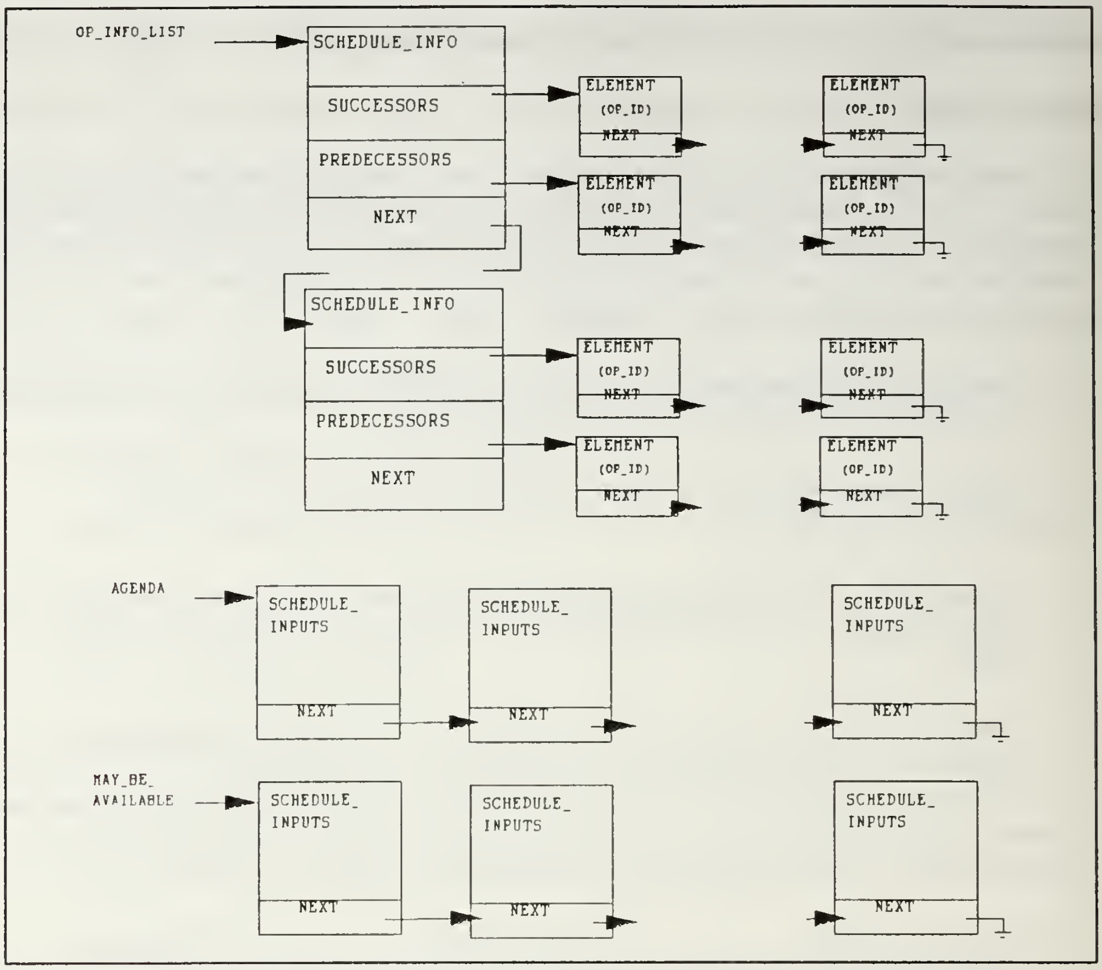


Figure 15 Linked List representations used in Algorithm 2 and Algorithm 3.

the operators THE_LOWER field in the SCHEDULE_INPUTS abstract data type is used. THE_START and THE_STOP fields are as being used in the first algorithm. Whenever an operator was selected from MAY_BE_AVAILABLE list and verified that all its predecessors are in AGENDA, it was taken out of the list. After it is processed, it was put back again in its order with new EST. The MAY_BE_AVAILABLE_LIST is kept in order because if all the predecessors are not in AGENDA during process 5, that operator is is skipped and the process is repeated for the next operator. In this ordered form, there is no necessity to look for the smallest EST in the list. The first operator always has the smallest EST. During the scheduling, if THE_STOP time of any operator is greater than the HARMONIC_BLOCK_LENGTH, then exception OVER_TIME is raised for that operator. This algorithm is not optimal as the branch and bound tree explained in Chapter III, but has the advantage that it is more compact in time and space.

G. IMPLEMENTATION OF "THE EARLIEST DEADLINE SCHEDULING ALGORITHM

The implementation of this algorithm is very similar to the "Earliest Start Scheduling Algorithm". Package utilization is the same as in the preceding algorithm. It also considers the precedence constraints among the operators. The only difference from the preceding algorithm is that the operators are selected according to their earliest deadlines instead of their earliest start times.The rest of this section describes in details, how the algorithm works with the procedures used in the OPERATOR_SCHEDULER.

1. "OPERATOR SCHEDULER" package

This package is shared with the other algorithms. It includes the procedure for the Earliest Deadline Scheduling Algorithm which is called SCHEDULE_WITH_EARLIEST_DEADLINE. The final output list(AGENDA) of this procedure is used by the procedure CREATE_STATIC_SCHEDULE for the final output. Procedure number 1 and functions number 3,4,6 shown on page 50 for the

Earliest Start Scheduling Algorithm are shared. The other procedures used by this algorithm are:

- procedure PROCESS_EDL_NODE
- procedure EDL_INSERT
- procedure EDL_INSERT_SUCCESSORS_OF_OPT
- procedure PROCESS_EDL_AGENDA

The two examples are given in Appendix A and Appendix B. The second example gives the same warning message as the others. Most of the criteria in this algorithm is the same as the Earliest Start Scheduling Algorithm. The major difference is the order of the MAY_BE_AVAILABLE_LIST which is ordered according to the earliest deadlines(EDL) of the operators. And this is considered during the scheduling process. The processes of this algorithm are explained in the following steps:

1. Find the operators which has no predecessors and put them all into the MAY_BE_AVAILABLE list in their orders according to their Earliest Deadlines(EDL). Since all these operators have different EDL, the order of the operators are important in here. Because of all the operators are in their orders according to their EDLs, we can pick the first one in the list. Since these have no predecessors, we do not need to check if the predecessors are in the AGENDA.

2. Select the first operator and put it into AGENDA list with a calculated THE_START and THE_STOP.

3. Define a new EDL for the selected operator and put it back into the MAY_BE_AVAILABLE list.

    If the operator has a FINISH_WITHIN in it then,

        EDL := EST + FINISH_WITHIN;

    otherwise;

        EDL := EST + THE_PERIOD;

4. Assign new EDL to each successor of the selected operator and insert them into the MAY_BE_AVAILABLE list in an order according to their EDL's.

5. Get the first operator with the smallest EDL in MAY_BE_AVAILABLE list and look if all its predecessors are in AGENDA. If the answer is no, then get the next operator and check the predecessors again. Repeat the process until the answer is yes. Then assign a new EDL for the selected operator and put it back i the MAY_BE_AVAILABLE_LIST in an order according to its EDL.

6. If any successor of the selected operator is not ALREADY in the MAY_BE_AVAILABLE list, assign a new EDL to that successor and insert into the MAY_BE_AVAILABLE list in its order.

7. Repeat the process 5 and 6 above until the EDL of the selected operator in MAY_BE_AVAILABLE list is greater or equal to the time interval(HBL). This is the stop condition and where "pointer = null".

H. SUMMARY

When the three algorithms are compared with eachother, The Earliest Start Scheduling Algorithm is more flexible and more efficient than the others. The way that it is presented in Chapter II uses a branch, exclude, and bound method. It searches all the branches in the tree one by one. But when the precedence relationships are considered, the disadvantage of this algorithm is the time complexity. Besides, it doesn't guarantee an optimal solution anymore. For these reasons, this algorithm is implemented with the Graph Structure. It was possible to construct the same structure as in Chapter two with Graph Structure, but it would be very hard to implement and we would need a very big storage capacity. Instead, the branches that we will not use are eliminated at the beginning, and this was the tradeoff between an optimal solution, and fast and easy implementation with less memory.

IV. DEVIATIONS FROM PREVIOUS WORK

There are some deviations from the previous implementation presented in Marlow [Ref.6] in this thesis. The assumptions made for the data requirements of the operators differentiates from the earlier assumptions to overcome some problems. The Graph Model is used as a basic structure instead of a N-ary tree structure for efficiency and simple process of the operators. The rest of the packages which are not implemented in Marlowe [Ref. 6] are completed. The "Exception_Handler" module included in this design is not the only level of exception handling, because the existance of some nonfatal exceptions raised during the execution do not require the programs to exit. Three different level exception handling exist in the implementation in this thesis.

A. ASSUMPTIONS

In Marlowe [Ref. 6: pp. 52-54], there was a problem mentioned in handling the non-time critical operators. The problem was how to separate the non-time critical operators whose data is required for a critical operator. Fig. 16 shows the situation. In this thesis, it is assumed that the operator between the two critical operators is always critical unless there is another path connecting the two critical operators. In this case, the output data of the non-time critical operator should be initialized. This handles the problem in separation of the non-time critical operators and so, since the OP_2 will not depend on the data of OP_4, OP_2 uses the new output data of OP_4 only when the dynamic scheduler executes the operator OP_4.

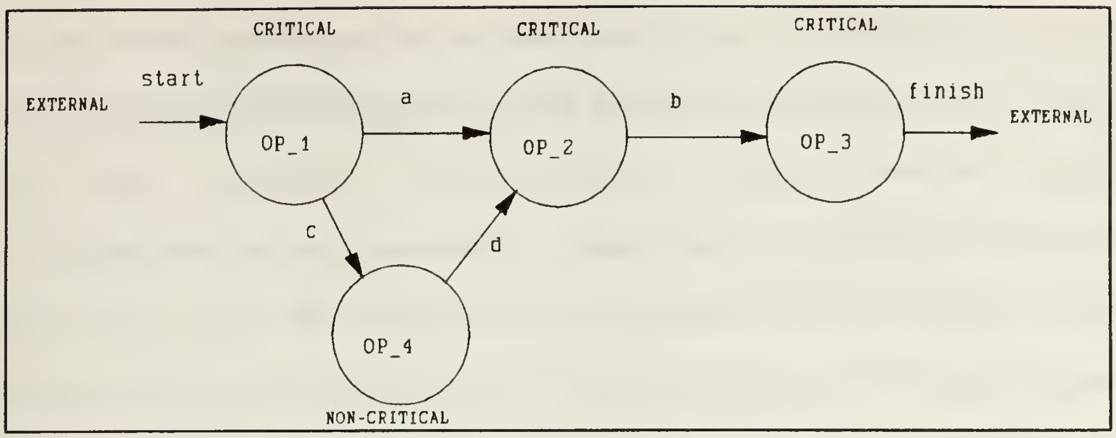


Figure 16 Example graph assumed for non-critical operators

B. DATA STRUCTURES

Although the abstract data types used for operator information and final scheduling is kept the same, the LINK_DATA abstract data type used for the link information is changed and some other data types are included for the other algorithms implemented. The LINK_DATA has a field called THE_LINK_MET; this field was not used during the implementation, but it is kept zero to show that we assume the time for the data flow for a single processor is zero. All the data structures are explained in details in Chapter III.

C. ARCHITECTURAL DESIGN

The architectural design in this implementation mostly looks like that presented in Marlowe [Ref. 6]. The Fig.4 and Fig.10 shows the differences in the two DFDL's. Since this implementation is the standalone static schedulers, an exception handler was needed, which is the same as the Debugger in CAPS. The second thing is the separation of the "PSDL_Reader" into "Preprocessor" and "Decomposer". The "Preprocessor" reads in the PSDL source file for the prototype being designed and produces a text file containing the information of the composite and atomic operators together. The resultant text file becomes the input to the module "Decomposer" which separates the atomic operator and link information, and does the validity checks between the composite and atomic operators. It produces a text file containing only the atomic operator and link information which becomes the input to the module "File_Processor". This separation makes the decomposition process easy and reduces the complexity.

D. EXCEPTION HANDLING

There are three different types of exception handling in this implementation. One of them is the outmost level exception handling which handles the major errors encountered during the execution. This is the same idea mentioned in Marlowe [Ref. 6]. Other level of exceptions are needed to run the static schedulers as standalone and to give warnings to the user without exiting the program. The details of the exception levels are given in Chapter III, Section D.

E. PACKAGE IMPLEMENTATION

The pseudo code listing given in Janson [Ref. 13] and the variable length string abstract data type, VSTRINGS, are utilized for the implementation of the first algorithm.

The OPERATOR_SCHEDULER package consists all the three static schedulers. The reason for not having different modules for every other static_scheduler is that all the static schedulers implemented here have the same time interval concept and share most of the procedures in this package.

In this implementation, the modules in the original 1st DFD are tried to have minimum change to keep the original design as simple as possible.

V. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY

This thesis provides three static schedulers which are the first complete implementations that support the Computer Aided Prototyping for Embedded Real-Time Systems. These schedulers can also be executed as standalone in the way that they are implemented.

Most of the algorithms written in the past do not address the problem of how to schedule tasks that have precedence constraints. Since the precedence constraints are considered in these algorithms, the graphical representation(directed graphs), and the notion of a base timeframe was used. None of the algorithms presented in this thesis gives an optimal solution to the problem of scheduling Hard Real-Time Systems with precedence constraints. But these schedulers are important in supporting the Execution Support System(ESS) within the framework of CAPS.

The contribution of this thesis to CAPS and Hard Real-Time Systems was the implemented static schedulers for non-preemptable, single processor systems. These static schedulers allow operators from any type of software system, even those with control based on data flow, to be scheduled in a way that meets all critical timing constraints.

B. CONCLUSIONS

With the implementation of these static schedulers in this thesis, the major part of the Static Scheduler in the ESS within CAPS is completed. These are integrated into the Execution Support System, with the simulation of "Decomposer". The new data structures like Graph Structure are introduced to the Static Scheduler. The Graph Model was very successful to capture the computational requirements of the Hard Real-Time Systems.

The schedulers are imported into the Execution Support System where "Decomposer" is simulated for the current STATIC_SCHEDULER. Since the composite operator information is not included in the graph data type, the names of the operators in STATIC_SCHEDULER output should start with the names of their composites to avoid the naming conflicts with the TRANSLATOR shown in Fig.3 The information of how these are related to eachother can be found in Palazzo [Ref. 32]. The driver program that runs the standalone static schedulers is adopted for the CAPS environment which is shown in Appendix F. Otherwise the schedulers were successfully used in CAPS.

All the programs in this thesis are implemented in Ada. Ada's modularization and generic package advantages with its exception handling mechanism were utilized to model the static schedulers for single processor. Even though Ada was very efficient for single processor environment, since it uses a FIFO queue for the parallel tasks, there would be a very big problem in the implementation of the schedulers for multiprocessor systems. When the tasks are queued during the parallel processing, we can not use any priority, or precedence relationship in the schedulers. This means Ada will need some changes for the implementation of optimal static schedulers.

Several areas for further research include the following:

- Implementation of the "Decomposer"
- Implementation of more efficient algorithms which give optimal solution to the scheduling problem
- Implementation of the static schedulers for preempted, multiprocessor systems
- To find a solution for the FIFO queue restriction for parallel tasking in Ada.

As soon as the "Decomposer" is completed and imported to the implementation, CAPS will not need any simulation for running the static schedulers. So the CAPS system will have a complete ESS running in its environment.

APPENDIX A. LINEAR GRAPH EXAMPLE

The following is the "atomic.info" file used as an input for the satic scheduleing algorithms in Figure 13.

```
ATOMIC
read_numbers
MET
10
PERIOD
20
ATOMIC
sort_numbers
MET
2
PERIOD
20
ATOMIC
write_numbers
MET
2
PERIOD
20
LINK
a
read_numbers
0
sort_numbers
LINK
b
sort_numbers
0
write_numbers
```

```
IMPLEMENTATION :
---------------
HARMONIC_BLOCK_LENGTH (HBL) = 20

OPERATOR_ID          MET         PERIOD
-----------         -----        ------
read_numbers         10            20
sort_numbers          2            20
write_numbers         2            20

1) FIRST ALGORITHM:
   ----------------

   PRECEDENCE_LIST  { read_numbers, sort_numbers, write_numbers }

   STATIC SCHEDULE:

   Message to the user:
   --------------------
   1- The total MET/PERIOD ratio sum of operators is greater than 0.5.
   2- Although a schedule may be possible, there is no guarantee that
      it will execute within the required timing constraints.

   OPERATOR_ID      START_TIME     END_TIME   FIRING_INTERVAL
   -----------      ----------     --------   ---------------
   read_numbers          0            10           (20,30)
   sort_numbers         10            12           (30,48)
   write_numbers        12            14           (32,50)

   ---------------------------------------------------------

   STOP CONDITION:  All firing intervals are greater than HBL in the last pass.
                    A feasible schedule found, READ "schedule.out" file.
```

```
2) SECOND ALGORITHM:  (Earliest_Start time Scheduling Algorithm)
   -----------------

   Message to the user:
   --------------------
   1- The total MET/PERIOD ratio sum of operators is greater than 0.5.
   2- Although a schedule may be possible, there is no guarantee that
      it will execute within the required timing constraints.

   SUCCESSORS                    :   PREDECESSORS                 :
   -----------------------------     ----------------------------
   read_numbers  [sort_numbers]      read_numbers   [-]
   sort_numbers  [write_numbers]     sort_numbers   [read_numbers]
   write_numbers [-]                 write_numbers  [sort_numbers]


         AGENDA                  :  MAY_BE_AVAILABLE                       :
         -------------------------  ------------------------------------------
STEP_1) [ ]                         [read_numbers] (end node)
                                         (EST:0)

STEP_2) [read_numbers]              [sort_numbers,read_numbers]
           START:0                       (EST:10)     (EST:20)
          FINISH:10

STEP_3) [read_numbers,sort_numbers]  [write_numbers,read_numbers,sort_numbers]
          START:0       START:10       (EST:12)       (EST:20)       (EST:30)
         FINISH:10     FINISH:12


        STATIC SCHEDULE:
        ----------------
STEP_4) [read_numbers,sort_numbers,write_numbers]
           START:0       START:10      START:12
          FINISH:10     FINISH:12     FINISH:14

                                    [read_numbers,sort_numbers,write_numbers]
                                      (EST:20)      (EST:30)       (EST:32)

        STOP CONDITION:  (All EST values are greater than HBL).
        A feasible schedule found, READ "ss.a" file.
```

```
3) THIRD ALGORITHM : (Earliest Deadline Scheduling Algorithm)
   -----------------

   Message to the user:
   --------------------
   1- The total MET/PERIOD ratio sum of operators is greater than 0.5.
   2- Although a schedule may be possible, there is no guarantee that
      it will execute within the required timing constraints.


   SUCCESSORS                    :  PREDECESSORS                :
   -----------------------------    ----------------------------
   read_numbers  [sort_numbers]     read_numbers  [-]
   sort_numbers  [write_numbers]    sort_numbers  [read_numbers]
   write_numbers [-]                write_numbers [sort_numbers]


         AGENDA                    :  MAY_BE_AVAILABLE                        :
         -------------------------    ------------------------------------------
STEP_1) [ ]                           [read_numbers] (end node)
                                           (EST:0)
                                           (EDL:20)

STEP_2) [read_numbers]                [sort_numbers,read_numbers]
           START:0                         (EST:10)    (EST:20)
          FINISH:10                        (EDL:30)    (EDL:40)

STEP_3) [read_numbers,sort_numbers]   [write_numbers,read_numbers,sort_numbers]
           START:0       START:10         (EST:12)      (EST:20)      (EST:30)
         FINISH:10      FINISH:12         (EDL:32)      (EDL:40)      (EDL:50)


         STATIC SCHEDULE:
         ----------------
STEP_4) [read_numbers,sort_numbers,write_numbers]
           START:0      START:10     START:12
          FINISH:10    FINISH:12    FINISH:14

                                      [read_numbers,sort_numbers,write_numbers]
                                        (EST:20)      (EST:30)      (EST:32)
                                        (EDL:40)      (EDL:50)      (EDL:52)

         STOP CONDITION:  (All EST values are greater than HBL).
         A feasible schedule found., READ "ss.a" file
```

The output "ss.a" file created as static schedule for the first algorithm :

```
------------------------------------------------------------------------
with TL; use TL;
with DS_PACKAGE; use DS_PACKAGE;
with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS;
with CALENDAR; use CALENDAR;
with TEXT_IO; use TEXT_IO;
procedure STATIC_SCHEDULE is
  write_numbers_TIMING_ERROR : exception;
  sort_numbers_TIMING_ERROR : exception;
  read_numbers_TIMING_ERROR : exception;
  task SCHEDULE is
    pragma priority (STATIC_SCHEDULE_PRIORITY);
  end SCHEDULE;

  task body SCHEDULE is
    PERIOD : constant := 20;
    read_numbers_STOP_TIME1 : constant :=  10.0;
    sort_numbers_STOP_TIME2 : constant :=  12.0;
    write_numbers_STOP_TIME3 : constant := 14.0;
    SLACK_TIME : duration;
    START_OF_PERIOD : time := clock;
begin
  loop
    begin
      read_numbers;
      SLACK_TIME := START_OF_PERIOD + read_numbers_STOP_TIME1 - CLOCK;
      if SLACK_TIME >= 0.0 then
        delay (SLACK_TIME);
      else
        raise read_numbers_TIMING_ERROR;
      end if;
      delay (START_OF_PERIOD +  10.0 - CLOCK);

      sort_numbers;
      SLACK_TIME := START_OF_PERIOD + sort_numbers_STOP_TIME2 - CLOCK;
      if SLACK_TIME >= 0.0 then
        delay (SLACK_TIME);
      else
        raise sort_numbers_TIMING_ERROR;
      end if;
      delay (START_OF_PERIOD +  12.0 - CLOCK);

      write_numbers;
      SLACK_TIME := START_OF_PERIOD + write_numbers_STOP_TIME3 - CLOCK;
      if SLACK_TIME >= 0.0 then
        delay (SLACK_TIME);
      else
        raise write_numbers_TIMING_ERROR;
      end if;
      START_OF_PERIOD := START_OF_PERIOD + PERIOD;
```

```
        delay (START_OF_PERIOD - clock);
        exception
          when write_numbers_TIMING_ERROR =>
            PUT_LINE("timing error from operator write_numbers");
            START_OF_PERIOD := clock;
          when sort_numbers_TIMING_ERROR =>
            PUT_LINE("timing error from operator sort_numbers");
            START_OF_PERIOD := clock;
          when read_numbers_TIMING_ERROR =>
            PUT_LINE("timing error from operator read_numbers");
            START_OF_PERIOD := clock;
        end;
      end loop;
    end SCHEDULE;

  begin
    null;
  end STATIC_SCHEDULE;
```

APPENDIX B.   ACYCLIC GRAPH EXAMPLE

The following is the "atomic.info" file used as an input for the Static Schedulers for Figure 16.

```
ATOMIC
OP_1
MET
1
PERIOD
12
ATOMIC
OP_2
MET
1
PERIOD
8
WITHIN
7
ATOMIC
OP_3
MET
1
MCP
8
MRT
12
ATOMIC
OP_4
MET
2
PERIOD
8
ATOMIC
OP_5
LINK
a1
OP_1
0
OP_2
LINK
a2
OP_1
0
OP_2
LINK
b
```

```
OP_2
0
OP_3
LINK
c
OP_2
0
OP_4
LINK
d
OP_3
0
OP_4
LINK
e
OP_1
0
OP_5
LINK
f
OP_5
0
OP_3
LINK
start
EXTERNAL
0
OPT_1
LINK
finish
OPT_4
0
EXTERNAL
```

```
IMPLEMENTATION :
---------------
HARMONIC_BLOCK_LENGTH (HBL) = 24

OPERATOR_ID         MET          PERIOD     FINISH_WITHIN
-----------        -----         ------     -------------
 OP_1               1              12              -
 OP_2               1               8              7
 OP_3               1               8(EQUIVALENT)  -
 OP_4               2               8              -

1) FIRST ALGORITHM: (Earliest Start Scheduling Algorithm)
   ------------------------------------------------------

   PRECEDENCE_LIST  { OP_1, OP_2, OP_3, OP_4 }

   STATIC SCHEDULE:

   Message to the user:
   --------------------
   1- The total MET/PERIOD ratio sum of operators is greater than 0.5.
   2- Although a schedule may be possible, there is no guarantee that
      it will execute within the required timing constraints.

   OPERATOR_ID      START_TIME    END_TIME    FIRING_INTERVAL
   -----------      ----------    --------    ---------------
   --------------
   First Process
   --------------
   OP_1                 0            1             (12,23)
   OP_2                 1            2            *(9,15)
   OP_3                 2            3             (10,17)
   OP_4                 3            5             (11,17)
   --------------
   Second Process
   --------------
   OP_1                12           13             (24,35)
   OP_2                13           14            *(17,23)
   OP_3                14           15             (18,25)
   OP_4                15           17             (19,25)
   ---------------------------------------------------
   OP_2                17           18            *(25,31)
   OP_3                18           19             (26,33)
   OP_4                19           21             (27,33)


   STOP CONDITION:  All firing intervals are greater than HBL in the last pass.
                    A feasible schedule found, READ "ss.a" file.
```

```
2) SECOND ALGORITHM: (Earliest Deadline Scheduling Algorithm)
   -------------------------------------------------------------

   Message to the user:
   --------------------
   1- The total MET/PERIOD ratio sum of operators is greater than 0.5.
   2- Although a schedule may be possible, there is no guarantee that
      it will execute within the required timing constraints.

   SUCCESSORS                     :  PREDECESSORS                 :
   ------------------------------    -----------------------------
   OP_1 [OP_2]                       OP_1 [-]
   OP_2 [OP_3,OP_4]                  OP_2 [OP_1]
   OP_3 [OP_4]                       OP_3 [OP_2]
   OP_4 [-]                          OP_4 [OP_2,OP_3]


         AGENDA                    :  MAY_BE_AVAILABLE                        :
         --------------------------   --------------------------------------------
STEP_1)  [ ]                            [OP_1] (end node)
                                        (EST:0)
STEP_2)  [OP_1]                         [ OP_2, OP_1]
         START:0                        (EST:1)(EST:12)
         FINISH:1
STEP_3) [...........,OP_2]              [ OP_3, OP_4,  OP_2,  OP_1 ]
                    START:1             (EST:2)(EST:2)(EST:9)(EST:12)
                    FINISH:2
STEP_4) [...........,OP_3]              [ OP_4,  OP_2,  OP_3,   OP_1 ]
                    START:2             (EST:2)(EST:9)(EST:10)(EST:12)
STEP_5) [......... .,OP_4]              [ OP_2,  OP_3,   OP_4,  OP_1 ]
                    START:3             (EST:9)(EST:10)(EST:11)(EST:12)
                    FINISH:5
STEP_6) [...........,OP_2]              [ OP_3,   OP_4,  OP_1,   OP_2 ]
                    START:9             (EST:10)(EST:11)(EST:12)(EST:17)
                    FINISH:10
STEP_7) [...........,OP_3]              [  OP_4,   OP_1,   OP_2,  OP_3 ]
                    START:10             (EST:11)(EST:12)(EST:17)(EST:18)
STEP_8) [...........,OP_4]              [  OP_1,   OP_2,   OP_3,   OP_4 ]
                    START:11             (EST:12)(EST:17)(EST:18)(EST:19)
                    FINISH:13
STEP_9) [...........,OP_1]              [ OP_2,   OP_3,   OP_4,   OP_1 ]
                    START:13             (EST:17)(EST:18)(EST:19)(EST:25)
                    FINISH:14
STEP_10) [...........,OP_2]              [ OP_3,   OP_4,   OP_1,   OP_2 ]
                    START:17             (EST:18)(EST:19)(EST:25)(EST:25)
                    FINISH:18
STEP_11) [...........,OP_3]              [ OP_4,   OP_1,   OP_2,   OP_3 ]
                    START:18             (EST:19)(EST:25)(EST:25)(EST:26)
                    FINISH:19
```

```
STEP_12) [...........,OP_4]              [ OP_1,    OP_2,    OP_3,    OP_4 ]
                   START:19              (EST:25) (EST:25) (EST:26) (EST:27)
                   FINISH:21

STOP CONDITION:  All EST values are greater than HBL in the last pass.

                 A feasible schedule found, READ "ss.a" file.
```

```
THE OUTPUT "ss.a" FILE CREATED AS STATIC SCHEDULE FOR THE FIRST ALGORITHM:
--------------------------------------------------------------------------
with TL; use TL;
with DS_PACKAGE; use DS_PACKAGE;
with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS;
with CALENDAR; use CALENDAR;
with TEXT_IO; use TEXT_IO;
procedure STATIC_SCHEDULE is
  OP_4_TIMING_ERROR : exception;
  OP_3_TIMING_ERROR : exception;
  OP_2_TIMING_ERROR : exception;
  OP_1_TIMING_ERROR : exception;
  task SCHEDULE is
    pragma priority (STATIC_SCHEDULE_PRIORITY);
  end SCHEDULE;

  task body SCHEDULE is
    PERIOD : constant := 24;
    OP_1_STOP_TIME1 : constant :=   1.0;
    OP_2_STOP_TIME2 : constant :=   2.0;
    OP_3_STOP_TIME3 : constant :=   3.0;
    OP_4_STOP_TIME4 : constant :=   5.0;
    OP_1_STOP_TIME5 : constant :=  13.0;
    OP_2_STOP_TIME6 : constant :=  14.0;
    OP_3_STOP_TIME7 : constant :=  15.0;
    OP_4_STOP_TIME8 : constant :=  17.0;
    OP_2_STOP_TIME9 : constant :=  18.0;
    OP_3_STOP_TIME10 : constant :=  19.0;
    OP_4_STOP_TIME11 : constant :=  21.0;
    SLACK_TIME : duration;
    START_OF_PERIOD : time := clock;
begin
  loop
    begin
      OP_1;
      SLACK_TIME := START_OF_PERIOD + OP_1_STOP_TIME1 - CLOCK;
      if SLACK_TIME >= 0.0 then
        delay (SLACK_TIME);
      else
        raise OP_1_TIMING_ERROR;
      end if;
      delay (START_OF_PERIOD +   1.0 - CLOCK);

      OP_2;
      SLACK_TIME := START_OF_PERIOD + OP_2_STOP_TIME2 - CLOCK;
      if SLACK_TIME >= 0.0 then
        delay (SLACK_TIME);
      else
        raise OP_2_TIMING_ERROR;
      end if;
      delay (START_OF_PERIOD +   2.0 - CLOCK);
```

```
OP_3;
SLACK_TIME := START_OF_PERIOD + OP_3_STOP_TIME3 - CLOCK;
if SLACK_TIME >= 0.0 then
  delay (SLACK_TIME);
else
  raise OP_3_TIMING_ERROR;
end if;
delay (START_OF_PERIOD +   3.0 - CLOCK);

OP_4;
SLACK_TIME := START_OF_PERIOD + OP_4_STOP_TIME4 - CLOCK;
if SLACK_TIME >= 0.0 then
  delay (SLACK_TIME);
else
  raise OP_4_TIMING_ERROR;
end if;
delay (START_OF_PERIOD +  12.0 - CLOCK);

OP_1;
SLACK_TIME := START_OF_PERIOD + OP_1_STOP_TIME5 - CLOCK;
if SLACK_TIME >= 0.0 then
  delay (SLACK_TIME);
else
  raise OP_1_TIMING_ERROR;
end if;
delay (START_OF_PERIOD +  13.0 - CLOCK);

OP_2;
SLACK_TIME := START_OF_PERIOD + OP_2_STOP_TIME6 - CLOCK;
if SLACK_TIME >= 0.0 then
  delay (SLACK_TIME);
else
  raise OP_2_TIMING_ERROR;
end if;
delay (START_OF_PERIOD +  14.0 - CLOCK);

OP_3;
SLACK_TIME := START_OF_PERIOD + OP_3_STOP_TIME7 - CLOCK;
if SLACK_TIME >= 0.0 then
  delay (SLACK_TIME);
else
  raise OP_3_TIMING_ERROR;
end if;
delay (START_OF_PERIOD +  15.0 - CLOCK);

OP_4;
SLACK_TIME := START_OF_PERIOD + OP_4_STOP_TIME8 - CLOCK;
if SLACK_TIME >= 0.0 then
  delay (SLACK_TIME);
else
  raise OP_4_TIMING_ERROR;
```

```
      end if;
      delay (START_OF_PERIOD +  17.0 - CLOCK);

      OP_2;
      SLACK_TIME := START_OF_PERIOD + OP_2_STOP_TIME9 - CLOCK;
      if SLACK_TIME >= 0.0 then
        delay (SLACK_TIME);
      else
        raise OP_2_TIMING_ERROR;
      end if;
      delay (START_OF_PERIOD +  18.0 - CLOCK);

      OP_3;
      SLACK_TIME := START_OF_PERIOD + OP_3_STOP_TIME10 - CLOCK;
      if SLACK_TIME >= 0.0 then
        delay (SLACK_TIME);
      else
        raise OP_3_TIMING_ERROR;
      end if;
      delay (START_OF_PERIOD +  19.0 - CLOCK);

      OP_4;
      SLACK_TIME := START_OF_PERIOD + OP_4_STOP_TIME11 - CLOCK;
      if SLACK_TIME >= 0.0 then
        delay (SLACK_TIME);
      else
        raise OP_4_TIMING_ERROR;
      end if;
      START_OF_PERIOD := START_OF_PERIOD + PERIOD;
      delay (START_OF_PERIOD - clock);
      exception
        when OP_4_TIMING_ERROR =>
          PUT_LINE("timing error from operator OP_4");
          START_OF_PERIOD := clock;
        when OP_3_TIMING_ERROR =>
          PUT_LINE("timing error from operator OP_3");
          START_OF_PERIOD := clock;
        when OP_2_TIMING_ERROR =>
          PUT_LINE("timing error from operator OP_2");
          START_OF_PERIOD := clock;
        when OP_1_TIMING_ERROR =>
          PUT_LINE("timing error from operator OP_1");
          START_OF_PERIOD := clock;
      end;
    end loop;
  end SCHEDULE;

begin
  null;
end STATIC_SCHEDULE;
```

APPENDIX C.   PREPROCESSOR and DECOMPOSER OUTPUTS

```
1) Preprocessor Output  :

LINEAGE
C1
read_numbers
sort_numbers
write_numbers
END_LINEAGE
C1
LINK
a
read_numbers
0
sort_numbers
LINK
b
sort_numbers
0
write_numbers
read_numbers
MET
10
PERIOD
20
sort_numbers
MET
2
PERIOD
20
write_numbers
MET
2
PERIOD
20
LINEAGE
read_numbers
ATOMIC
END_LINEAGE
read_numbers
read_numbers
LINEAGE
sort_numbers
ATOMIC
END_LINEAGE
```

```
sort_numbers
sort_numbers
LINEAGE
write_numbers
ATOMIC
END_LINEAGE
write_numbers
write_numbers
```

2) Decomposer Output :

```
ATOMIC
read_numbers
MET
10
PERIOD
20
ATOMIC
sort_numbers
MET
2
PERIOD
20
ATOMIC
write_numbers
MET
2
PERIOD
20
LINK
a
read_numbers
0
sort_numbers
LINK
b
sort_numbers
0
write_numbers
```

APPENDIX D. PROGRAM DOCUMENTATION

1. STANDALONE MENU DRIVEN VERSION AS IMPLEMENTED IN THIS THESIS

```
preprocessor    - generates the text file used by decomposer
        (not implemented )
decomposer_b.a    - validates and decomposes output of preprocessor
      (not implemented yet)
decomposer_s.a    - validates and decomposes output of preprocessor
      (not implemented yet)
driver.a          - interface for standalone static scheduler
e_handler_b.a           - exception routines used by driver
e_handler_s.a           - exception routines used by driver
files.a                 - global types and declarations for all ss programs
fp_b.a                  - file processor
fp_s.a                  - file processor
graphs_b.a        - generic type graph structure
graphs_s.a        - generic type graph structure
hbb_b.a                 - harmonic block builder
hbb_s.a                 - harmonic block builder
scheduler_b.a           - operators_scheduler (scheduling algorithms)
scheduler_s.a           - operators_scheduler (scheduling algorithms)
sequence_b.a            - generic type list structure
sequence_s.a            - generic type list structure
static_scheduler* - executable static_scheduler
t_sort_b.a        - topological sorter
t_sort_s.a        - topological sorter

static_scheduler is compiled by:
      a.make static_scheduler -f *.a -o static_scheduler
      ( where *.a uses all files listed above which have a .a suffix )

static_scheduler is executed by the command line equivalent
      static_scheduler (expects to read an input file "atomic.info")

Dependencies:

      files.a is dependent upon:
            vstrings
            sequences
            graphs

      decomposer_b.a, decomposer_s.a, e_handler_b.a, e_handler_s.a,
      fp_b.a, fp_s.a, hbb_b.a, hbb_s.a, scheduler_b.a, scheduler_s.a,
      t_sort_b.a, t_sort_s.a are all dependent upon:
            files (files.a)

      driver.a is dependent upon
            decomposer (decomposer_b.a, decomposer_s.a)
                 (atomic.info file is given to the system)
            exception_handler (e_handler_b.a, e_handler_s.a)
            file_processor (fp_b.a, fp_s.a)
            harmonic_block_builder  (hbb_b.a, hbb_s.a)
            operator_scheduler (scheduler_b.a, scheduler_s.a)
            topological_sorter (t_sort_b.a, t_sort_s.a)

since decomposer is not implemented yet, atomic.info file is given.
File_processor reads atomic.info
File_processor creates non_crits.a
Operator_scheduler creates ss.a
```

2. DOCUMENTATION FOR THE COMPLETE DESIGN AS IT WILL BE USED IN CAPS:

```
decomposer_b.a    - validates and decomposes output of preprocessor
(not implemented yet)
decomposer_s.a    - validates and decomposes output of preprocessor
(not implemented yet)
driver.a          - interface for standalone static scheduler
e_handler_b.a           - exception routines used by driver
e_handler_s.a           - exception routines used by driver
files.a                 - global types and declarations for all ss programs
fp_b.a                  - file processor
fp_s.a                  - file processor
graphs_b.a        - generic type graph structure
graphs_s.a        - generic type graph structure
hbb_b.a                 - harmonic block builder
hbb_s.a                 - harmonic block builder
kc                - script to compile static scheduler preprocess pre_ss.k
pre_ss*                 - executable preprocessor
pre_ss.k          - kodiyacc specifications for preprocessor
scheduler_b.a           - operators_scheduler (scheduling algorithms)
scheduler_s.a           - operators_scheduler (scheduling algorithms)
sequence_b.a            - generic type list structure
sequence_s.a            - generic type list structure
static_scheduler* - executable static_scheduler
t_sort_b.a        - topological sorter
t_sort_s.a        - topological sorter

The caps static scheduler consists of two executable modules.
pre_ss is compiled by:
      kc pre_ss.k -o pre_ss

pre_ss is executed by the command line equivalent
      pre_ss <filename> -o operator.info
```

```
static_scheduler is compiled by:
      a.make static_scheduler -f *.a -o static_scheduler
      ( where *.a uses all files listed above which have a .a suffix )

static_scheduler is executed by the command line equivalent
      static_scheduler (expects to read an input file "atomic.info")

Dependencies:

      files.a is dependent upon:
            vstrings
            sequences
            graphs

      decomposer_b.a, decomposer_s.a, e_handler_b.a, e_handler_s.a,
      fp_b.a, fp_s.a, hbb_b.a, hbb_s.a, scheduler_b.a, scheduler_s.a,
      t_sort_b.a, t_sort_s.a are all dependent upon:
            files (files.a)

      driver.a is dependent upon
            decomposer (decomposer_b.a, decomposer_s.a)
            exception_handler (e_handler_b.a, e_handler_s.a)
            file_processor (fp_b.a, fp_s.a)
            harmonic_block_builder  (hbb_b.a, hbb_s.a)
            operator_scheduler (scheduler_b.a, scheduler_s.a)
            topological_sorter (t_sort_b.a, t_sort_s.a)

pre_ss creates operator.info
decomposer reads operator.info and creates atomic.info
File_processor reads atomic.info
File_processor creates non_crits.a
Operator_scheduler creates ss.a
```

APPENDIX E. IMPLEMENTATION OF THE STATIC SCHEDULING ALGORITHM

This appendix contains the entire implementation for the Static Scheduler.

```
--==========================================================================
-- SEQUENCES - this is a generic package used by the FILES and GRAPHS package
--             to generate Linked Lists.
--==========================================================================
with FILES; use FILES;
package OPERATOR_SCHEDULER is

  procedure TEST_DATA (INPUT_LIST            : in DIGRAPH.V_LISTS.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER);

  procedure SCHEDULE_INITIAL_SET (PRECEDENCE_LIST : in DIGRAPH.V_LISTS.LIST;
                       THE_SCHEDULE_INPUTS   : in out SCHEDULE_INPUTS_LIST.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER;
                       STOP_TIME             : in out INTEGER);

  procedure SCHEDULE_REST_OF_BLOCK(PRECEDENCE_LIST : in DIGRAPH.V_LISTS.LIST;
                       THE_SCHEDULE_INPUTS   : in out SCHEDULE_INPUTS_LIST.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER;
                       STOP_TIME             : in INTEGER);

  procedure SCHEDULE_WITH_EARLIEST_START (THE_GRAPH : in DIGRAPH.GRAPH;
                       AGENDA                : in out SCHEDULE_INPUTS_LIST.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER);

  procedure SCHEDULE_WITH_EARLIEST_DEADLINE (THE_GRAPH : in DIGRAPH.GRAPH;
                       AGENDA                : in out SCHEDULE_INPUTS_LIST.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER);

  procedure CREATE_STATIC_SCHEDULE (THE_GRAPH    : in DIGRAPH.GRAPH;
                           THE_SCHEDULE_INPUTS   : in SCHEDULE_INPUTS_LIST.LIST;
                           HARMONIC_BLOCK_LENGTH : in INTEGER);

  MISSED_DEADLINE  : exception;
  OVER_TIME        : exception;
  MISSED_OPERATOR  : exception;

end OPERATOR_SCHEDULER;
-------------------------------
with UNCHECKED_DEALLOCATION;

package body SEQUENCES is

  procedure FREE is new UNCHECKED_DEALLOCATION(NODE, LIST);
```

```
function NON_EMPTY(L : in LIST) return BOOLEAN is
begin
  if L = null then
    return FALSE;
  else
    return TRUE;
  end if;
end NON_EMPTY;

procedure NEXT(L : in out LIST) is
begin
  if L /= null then
    L := L.NEXT;
  end if;
end NEXT;

function LOOK4(X : in ITEM; L : in LIST) return LIST is
  L1 : LIST := L;
begin
  while NON_EMPTY(L1) loop
    if L1.ELEMENT = X then
      return L1;
    end if;
    NEXT(L1);
  end loop;
  return null;
end LOOK4;

procedure ADD(X : in ITEM; L : in out LIST) is
-- ITEM IS ADDED TO THE HEAD OF THE LIST
  T : LIST := new NODE;
begin
  T.ELEMENT := X;
  T.NEXT := L;
  L := T;
end ADD;

function SUBSEQUENCE(L1 : in LIST; L2 : in LIST) return BOOLEAN is
  L : LIST := L1;
begin
  while NON_EMPTY(L) loop
    if not MEMBER(VALUE(L), L2) then
      return FALSE;
    end if;
    NEXT(L);
  end loop;
  return TRUE;
end SUBSEQUENCE;

function EQUAL(L1 : in LIST; L2 : in LIST) return BOOLEAN is
```

```
begin
  return (SUBSEQUENCE(L1, L2) and SUBSEQUENCE(L2, L1));
end EQUAL;

procedure EMPTY(L : out LIST) is
begin
  L := null;
end EMPTY;

function MEMBER(X : in ITEM; L : in LIST) return BOOLEAN is
begin
  if LOOK4(X, L) /= null then
    return TRUE;
  else
    return FALSE;
  end if;
end MEMBER;

procedure REMOVE(X : in ITEM; L : in out LIST) is
  CURR : LIST := L;
  PREV : LIST := null;
  TEMP : LIST := null;
begin
  while NON_EMPTY(CURR) loop
    if VALUE(CURR) = X then
      TEMP := CURR;
      NEXT(CURR);
      FREE(TEMP);
      if PREV /= null then
        PREV.NEXT := CURR;
      else
        L := CURR;
      end if;
    else
      PREV := CURR;
      NEXT(CURR);
    end if;
  end loop;
end REMOVE;

procedure LIST_REVERSE(L1 : in LIST; L2 : in out LIST) is
  L : LIST := L1;
begin
  EMPTY(L2);
  while NON_EMPTY(L) loop
    ADD(VALUE(L), L2);
    NEXT(L);
  end loop;
end LIST_REVERSE;

procedure DUPLICATE(L1 : in LIST; L2 : in out LIST) is
```

```
    TEMP : LIST;
    L    : LIST := L1;
  begin
    EMPTY(L2);
    while NON_EMPTY(L) loop
      ADD(VALUE(L), TEMP);
      NEXT(L);
    end loop;
    LIST_REVERSE(TEMP, L2);
  end DUPLICATE;

  function VALUE(L : in LIST) return ITEM is
  begin
    if NON_EMPTY(L) then
      return L.ELEMENT;
    else
      raise BAD_VALUE;
    end if;
  end VALUE;

end SEQUENCES;
```

```
--=============================================================================
-- GRAPHS - a generic package used by the FILES package to generate
--          Graph Structure.
--=============================================================================
with SEQUENCES;
with VSTRINGS;

generic
  type VERTEX is private;

package GRAPHS is

  package V_LISTS is new SEQUENCES(VERTEX);
  use V_LISTS;

  package V_STRING is new VSTRINGS(80);
  use V_STRING;

  subtype DATA_STREAM is VSTRING;
  subtype MET is NATURAL;

  type LINK_DATA is
    record
      THE_DATA_STREAM   : DATA_STREAM;
      THE_FIRST_OP_ID   : V_LISTS.LIST;
      THE_LINK_MET      : MET := 0;
      THE_SECOND_OP_ID  : V_LISTS.LIST;
    end record;

  package E_LISTS is new SEQUENCES(LINK_DATA);
  use E_LISTS;

  type GRAPH is
    record
      VERTICES : V_LISTS.LIST;
      LINKS    : E_LISTS.LIST;
    end record;

  function EQUAL_GRAPHS(G1 : in GRAPH; G2 : in GRAPH) return BOOLEAN;

  procedure EMPTY(G : out GRAPH);

  function IS_NODE(X : in VERTEX; G : GRAPH) return BOOLEAN;

  function IS_LINK(X : in VERTEX; Y : in VERTEX;
                                  G : in GRAPH) return BOOLEAN;

  procedure ADD(X : in VERTEX; G : in out GRAPH);

  procedure ADD(L : in LINK_DATA; G : in out GRAPH);
  procedure REMOVE(X : in VERTEX; G : in out GRAPH);

  procedure REMOVE(X : in VERTEX; Y : in VERTEX; G : in out GRAPH);

  procedure SCAN_NODES(G : in GRAPH; S : in out V_LISTS.LIST);

  procedure SCAN_PARENTS(X : in VERTEX; G : in GRAPH;
                                        S : in out V_LISTS.LIST);

  procedure SCAN_CHILDREN(X : in VERTEX; G : in GRAPH;
                                         S : in out V_LISTS.LIST);

  procedure DUPLICATE(G1 : in GRAPH; G2 : in out GRAPH);

  procedure T_SORT(G : in GRAPH; S : in out V_LISTS.LIST);

end GRAPHS;
------------------------------------
with UNCHECKED_DEALLOCATION;

package body GRAPHS is

  procedure FREE is new UNCHECKED_DEALLOCATION(E_LISTS.NODE, E_LISTS.LIST);

  function EQUAL_GRAPHS(G1 : in GRAPH; G2 : in GRAPH) return BOOLEAN is

    function SUB_SET(G1 : in GRAPH; G2 : in GRAPH) return BOOLEAN is
      L1 : V_LISTS.LIST := G1.VERTICES;
      L2 : E_LISTS.LIST := G1.LINKS;
    begin
      if not SUBSEQUENCE(L1, G2.VERTICES) then
        return FALSE;
      end if;
      while NON_EMPTY(L2) loop
        if not IS_LINK(VALUE(VALUE(L2).THE_FIRST_OP_ID),
                       VALUE(VALUE(L2).THE_SECOND_OP_ID), G2) then
          return FALSE;
        end if;
        NEXT(L2);
      end loop;
      return TRUE;
    end SUB_SET;
  begin
    -- equal_graphs
    return (SUB_SET(G1, G2) and SUB_SET(G2, G1));
  end EQUAL_GRAPHS;

  procedure EMPTY(G : out GRAPH) is
  begin
    EMPTY(G.VERTICES);
    EMPTY(G.LINKS);
```

```
end EMPTY;

function IS_NODE(X : in VERTEX; G : GRAPH) return BOOLEAN is
begin
  if LOOK4(X, G.VERTICES) /= null then
    return TRUE;
  else
    return FALSE;
  end if;
end IS_NODE;

function IS_LINK(X : in VERTEX; Y : in VERTEX; G : in GRAPH) return BOOLEAN is
  L : E_LISTS.LIST := G.LINKS;
begin
  while L /= null loop
    if VALUE(VALUE(L).THE_FIRST_OP_ID) = X and
             VALUE(VALUE(L).THE_SECOND_OP_ID) = Y then
      return TRUE;
    end if;
    L := L.NEXT;
  end loop;
  return FALSE;
end IS_LINK;

procedure ADD(X : in VERTEX; G : in out GRAPH) is
begin
  ADD(X, G.VERTICES);
end ADD;

procedure ADD(L : in LINK_DATA; G : in out GRAPH) is
begin
  if LOOK4(L.THE_FIRST_OP_ID.ELEMENT, G.VERTICES) /= null and
     LOOK4(L.THE_SECOND_OP_ID.ELEMENT, G.VERTICES) /= null then

    ADD(L, G.LINKS);

  end if;
end ADD;

procedure REMOVE(X : in VERTEX; G : in out GRAPH) is
  S : V_LISTS.LIST;
  L : V_LISTS.LIST;
  PREV : V_LISTS.LIST := null;
begin
  SCAN_CHILDREN(X, G, S);
  while NON_EMPTY(S) loop
    REMOVE(X, VALUE(S), G);
    NEXT(S);
  end loop;
  SCAN_PARENTS(X, G, S);
  while NON_EMPTY(S) loop
```

```
      REMOVE(VALUE(S), X, G);
      NEXT(S);
    end loop;
    REMOVE(X, G.VERTICES);
  end REMOVE;

  procedure REMOVE(X : in VERTEX; Y : in VERTEX; G : in out GRAPH) is
    L : E_LISTS.LIST := G.LINKS;
    PREV : E_LISTS.LIST := null;
    TEMP : E_LISTS.LIST := null;
  begin
    while NON_EMPTY(L) loop
      if VALUE(VALUE(L).THE_FIRST_OP_ID) = X and
               VALUE(VALUE(L).THE_SECOND_OP_ID) = Y then
        TEMP := L;
        NEXT(L);
        FREE(TEMP);
        if PREV /= null then
          PREV.NEXT := L;
        else
          G.LINKS := L;
        end if;
      else
        PREV := L;
        NEXT(L);
      end if;
    end loop;
  end REMOVE;

  procedure SCAN_NODES(G : in GRAPH; S : in out V_LISTS.LIST) is
    L : V_LISTS.LIST := G.VERTICES;
  begin
    EMPTY(S);
    while NON_EMPTY(L) loop
      ADD(VALUE(L), S);
      NEXT(L);
    end loop;
  end SCAN_NODES;

  procedure SCAN_PARENTS(X : in VERTEX; G : in GRAPH;
                                        S : in out V_LISTS.LIST) is
    L : E_LISTS.LIST := G.LINKS;

  begin
    EMPTY(S);
    while NON_EMPTY(L) loop
      if VALUE(VALUE(L).THE_SECOND_OP_ID) = X then
        ADD(VALUE(VALUE(L).THE_FIRST_OP_ID), S);
      end if;
      NEXT(L);
    end loop;
  end SCAN_PARENTS;

  procedure SCAN_CHILDREN(X : in VERTEX; G : in GRAPH;
                                         S : in out V_LISTS.LIST) is
    L : E_LISTS.LIST := G.LINKS;
  begin
    EMPTY(S);
    while NON_EMPTY(L) loop
      if VALUE(VALUE(L).THE_FIRST_OP_ID) = X then
        ADD(VALUE(VALUE(L).THE_SECOND_OP_ID), S);
      end if;
      NEXT(L);
    end loop;
  end SCAN_CHILDREN;

  procedure DUPLICATE(G1 : in GRAPH; G2 : in out GRAPH) is
  begin
    DUPLICATE(G1.VERTICES, G2.VERTICES);
    DUPLICATE(G1.LINKS, G2.LINKS);
  end DUPLICATE;

  procedure T_SORT(G : in GRAPH; S : in out V_LISTS.LIST) is
    G1 : GRAPH;
    T, L, P : V_LISTS.LIST;
  begin
    EMPTY(T);
    DUPLICATE(G, G1);
    SCAN_NODES(G1, L);
    while NON_EMPTY(L) loop
      SCAN_PARENTS(VALUE(L), G1, P);
      if not NON_EMPTY(P) then
        ADD(VALUE(L), T);
        REMOVE(VALUE(L), G1);
        SCAN_NODES(G1, L);
      else
        NEXT(L);
      end if;
    end loop;
    SCAN_NODES(G1, L);
    if NON_EMPTY(L) then
      EMPTY(S);
    else
      LIST_REVERSE(T, S);
    end if;
  end T_SORT;
end GRAPHS;
```

```
--=============================================================================
-- VSTRINGS - "vstrng_s.a, vstrng_b.a"; this is a generic package used within
--             the Static Scheduler for variable length string types.
--=============================================================================
with TEXT_IO; use TEXT_IO;
generic
  LAST : NATURAL;
package VSTRINGS is

  subtype STRINDEX is NATURAL;
  FIRST : constant STRINDEX := STRINDEX'FIRST + 1;
  type VSTRING is private;
  NUL : constant VSTRING;

-- Attributes of a VSTRING

  function LEN(FROM : VSTRING) return STRINDEX;
  function MAX(FROM : VSTRING) return STRINDEX;
  function STR(FROM : VSTRING) return STRING;
  function CHAR(FROM: VSTRING; POSITION : STRINDEX := FIRST)
                return CHARACTER;

-- Comparisons

  function "<" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
  function ">" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
  function "<=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
  function ">=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
  function EQUAL (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
  function NOTEQUAL (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;


-- Input/Output

  procedure PUT(FILE : in FILE_TYPE; ITEM : in VSTRING);
  procedure PUT(ITEM : in VSTRING);

  procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in VSTRING);
  procedure PUT_LINE(ITEM : in VSTRING);

  procedure GET(FILE : in FILE_TYPE; ITEM : out VSTRING;
                LENGTH : in STRINDEX := LAST);
  procedure GET(ITEM : out VSTRING; LENGTH : in STRINDEX := LAST);

  procedure GET_LINE(FILE : in FILE_TYPE; ITEM : in out VSTRING);
  procedure GET_LINE(ITEM : in out VSTRING);

-- Extraction

  function SLICE(FROM: VSTRING; FRONT, BACK : STRINDEX) return VSTRING;
  function SUBSTR(FROM: VSTRING; START, LENGTH: STRINDEX) return VSTRING;
  function DELETE(FROM: VSTRING; FRONT, BACK : STRINDEX) return VSTRING;

-- Editing

  function INSERT(TARGET: VSTRING; ITEM: VSTRING;
                  POSITION: STRINDEX := FIRST) return VSTRING;
  function INSERT(TARGET: VSTRING; ITEM: STRING;
                  POSITION: STRINDEX := FIRST) return VSTRING;
  function INSERT(TARGET: VSTRING; ITEM: CHARACTER;
                  POSITION: STRINDEX := FIRST) return VSTRING;

  function APPEND(TARGET: VSTRING; ITEM: VSTRING; POSITION: STRINDEX)
                  return VSTRING;
  function APPEND(TARGET: VSTRING; ITEM: STRING; POSITION: STRINDEX)
                  return VSTRING;
  function APPEND(TARGET: VSTRING; ITEM: CHARACTER; POSITION: STRINDEX)
                  return VSTRING;

  function APPEND(TARGET: VSTRING; ITEM: VSTRING) return VSTRING;
  function APPEND(TARGET: VSTRING; ITEM: STRING) return VSTRING;
  function APPEND(TARGET: VSTRING; ITEM: CHARACTER) return VSTRING;

  function REPLACE(TARGET: VSTRING; ITEM: VSTRING;
                   POSITION: STRINDEX := FIRST) return VSTRING;
  function REPLACE(TARGET: VSTRING; ITEM: STRING;
                   POSITION: STRINDEX := FIRST) return VSTRING;
  function REPLACE(TARGET: VSTRING; ITEM: CHARACTER;
                   POSITION: STRINDEX := FIRST) return VSTRING;

-- Concatenation

  function "&" (LEFT: VSTRING; RIGHT : VSTRING) return VSTRING;
  function "&" (LEFT: VSTRING; RIGHT : STRING) return VSTRING;
  function "&" (LEFT: VSTRING; RIGHT : CHARACTER) return VSTRING;
  function "&" (LEFT: STRING; RIGHT : VSTRING) return VSTRING;
  function "&" (LEFT: CHARACTER; RIGHT : VSTRING) return VSTRING;

-- Determine the position of a substring

  function INDEX(WHOLE: VSTRING; PART: VSTRING; OCCURRENCE : NATURAL := 1)
                 return STRINDEX;
  function INDEX(WHOLE : VSTRING; PART : STRING; OCCURRENCE : NATURAL := 1)
                 return STRINDEX;
  function INDEX(WHOLE : VSTRING; PART : CHARACTER; OCCURRENCE : NATURAL := 1)
                 return STRINDEX;


  function RINDEX(WHOLE: VSTRING; PART: VSTRING; OCCURRENCE : NATURAL := 1)
                 return STRINDEX;
  function RINDEX(WHOLE : VSTRING; PART : STRING; OCCURRENCE : NATURAL := 1)
                 return STRINDEX;
  function RINDEX(WHOLE : VSTRING; PART : CHARACTER; OCCURRENCE : NATURAL := 1)
                return STRINDEX;

-- Conversion from other associated types

  function VSTR(FROM : STRING) return VSTRING;
  function VSTR(FROM : CHARACTER) return VSTRING;
  function "+" (FROM : STRING) return VSTRING;
  function "+" (FROM : CHARACTER) return VSTRING;

  generic
    type FROM is private;
    type TO is private;
    with function STR(X : FROM) return STRING is <>;
    with function VSTR(Y : STRING) return TO is <>;
   function CONVERT(X : FROM) return TO;

  private
    type VSTRING is
      record
        LEN : STRINDEX := STRINDEX'FIRST;
        VALUE : STRING(FIRST .. LAST) := (others => ASCII.NUL);
      end record;

    NUL : constant VSTRING := (STRINDEX'FIRST, (others => ASCII.NUL));
end VSTRINGS;
----------------------------
package body VSTRINGS is

  -- local declarations

  FILL_CHAR : constant CHARACTER := ASCII.NUL;

  procedure FORMAT(THE_STRING: in out VSTRING;
                   OLDLEN     : in STRINDEX:=LAST) is
    -- fill the string with FILL_CHAR to null out old values

    begin -- FORMAT (Local Procedure)
      THE_STRING.VALUE(THE_STRING.LEN + 1 .. OLDLEN) :=
                                         (others => FILL_CHAR);
    end FORMAT;


  -- bodies of visible operations

  function LEN(FROM : VSTRING) return STRINDEX is

    begin -- LEN
      return(FROM.LEN);
    end LEN;
```

```
function MAX(FROM : VSTRING) return STRINDEX is
  begin -- MAX
    return(LAST);
  end MAX;


function STR(FROM : VSTRING) return STRING is
  begin -- STR
    return(FROM.VALUE(FIRST .. FROM.LEN));
  end STR;


function CHAR(FROM : VSTRING; POSITION : STRINDEX := FIRST)
               return CHARACTER is

  begin -- CHAR
    if POSITION not in FIRST .. FROM.LEN
      then raise CONSTRAINT_ERROR;
     end if;
    return(FROM.VALUE(POSITION));
  end CHAR;


function "<" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
  begin -- "<"
    return(LEFT.VALUE < RIGHT.VALUE);
  end "<";


function ">" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
  begin -- ">"
    return(LEFT.VALUE > RIGHT.VALUE);
  end ">";


function "<=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
  begin -- "<="
    return(LEFT.VALUE <= RIGHT.VALUE);
  end "<=";


function ">=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
  begin -- ">="
    return(LEFT.VALUE >= RIGHT.VALUE);
  end ">=";

function equal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
  begin -- equal
    return(LEFT.VALUE = RIGHT.VALUE);
  end equal;
```

```
function notequal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
  begin -- notequal
    return(LEFT.VALUE /= RIGHT.VALUE);
  end notequal;


procedure PUT(FILE : in FILE_TYPE; ITEM : in VSTRING) is
  begin -- PUT
    PUT(FILE, ITEM.VALUE(FIRST .. ITEM.LEN));
  end PUT;

procedure PUT(ITEM : in VSTRING) is
  begin -- PUT
    PUT(ITEM.VALUE(FIRST .. ITEM.LEN));
  end PUT;


procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in VSTRING) is
  begin -- PUT_LINE
    PUT_LINE(FILE, ITEM.VALUE(FIRST .. ITEM.LEN));
  end PUT_LINE;

procedure PUT_LINE(ITEM : in VSTRING) is
  begin -- PUT_LINE
    PUT_LINE(ITEM.VALUE(FIRST .. ITEM.LEN));
  end PUT_LINE;


procedure GET(FILE : in FILE_TYPE; ITEM : out VSTRING;
              LENGTH : in STRINDEX := LAST) is
  begin -- GET
    if LENGTH not in FIRST .. LAST
      then raise CONSTRAINT_ERROR;
     end if;

    ITEM := NUL;
    for INDEX in FIRST .. LENGTH loop
      GET(FILE, ITEM.VALUE(INDEX));
      ITEM.LEN := INDEX;
     end loop;
  end GET;

procedure GET(ITEM : out VSTRING; LENGTH : in STRINDEX := LAST) is
  begin -- GET
    if LENGTH not in FIRST .. LAST
      then raise CONSTRAINT_ERROR;
     end if;

    ITEM := NUL;
    for INDEX in FIRST .. LENGTH loop
```

```
      GET(ITEM.VALUE(INDEX));
      ITEM.LEN := INDEX;
     end loop;
  end GET;


procedure GET_LINE(FILE : in FILE_TYPE; ITEM : in out VSTRING) is

  OLDLEN : constant STRINDEX := ITEM.LEN;

  begin -- GET_LINE
    GET_LINE(FILE, ITEM.VALUE, ITEM.LEN);
    FORMAT(ITEM, OLDLEN);
  end GET_LINE;

procedure GET_LINE(ITEM : in out VSTRING) is

  OLDLEN : constant STRINDEX := ITEM.LEN;

  begin -- GET_LINE
    GET_LINE(ITEM.VALUE, ITEM.LEN);
    FORMAT(ITEM, OLDLEN);
  end GET_LINE;


function SLICE(FROM : VSTRING; FRONT, BACK : STRINDEX) return VSTRING is

  begin -- SLICE
    if ((FRONT not in FIRST .. FROM.LEN) or else
       (BACK not in FIRST .. FROM.LEN)) and then FRONT <= BACK
      then raise CONSTRAINT_ERROR;
     end if;

    return(Vstr(FROM.VALUE(FRONT .. BACK)));
  end SLICE;


function SUBSTR(FROM : VSTRING; START, LENGTH : STRINDEX) return VSTRING is

  begin -- SUBSTR
    if (START not in FIRST .. FROM.LEN) or else
       ((START + LENGTH - 1 not in FIRST .. FROM.LEN)
        and then (LENGTH > 0))
      then raise CONSTRAINT_ERROR;
     end if;

    return(Vstr(FROM.VALUE(START .. START + LENGTH -1)));
  end SUBSTR;


function DELETE(FROM : VSTRING; FRONT, BACK : STRINDEX) return VSTRING is
```

```
  TEMP : VSTRING := FROM;

  begin -- DELETE
    if ((FRONT not in FIRST .. FROM.LEN) or else
       (BACK not in FIRST .. FROM.LEN)) and then FRONT <= BACK
      then raise CONSTRAINT_ERROR;
     end if;

    if FRONT > BACK then return(FROM); end if;
    TEMP.LEN := FROM.LEN - (BACK - FRONT) - 1;

    TEMP.VALUE(FRONT .. TEMP.LEN) := FROM.VALUE(BACK + 1 .. FROM.LEN);
    FORMAT(TEMP, FROM.LEN);
    return(TEMP);
  end DELETE;


function INSERT(TARGET: VSTRING; ITEM: VSTRING;
                POSITION : STRINDEX := FIRST) return VSTRING is

  TEMP : VSTRING;

  begin -- INSERT
    if POSITION not in FIRST .. TARGET.LEN
      then raise CONSTRAINT_ERROR;
     end if;

    if TARGET.LEN + ITEM.LEN > LAST
      then raise CONSTRAINT_ERROR;
      else TEMP.LEN := TARGET.LEN + ITEM.LEN;
     end if;

    TEMP.VALUE(FIRST .. POSITION - 1) := TARGET.VALUE(FIRST .. POSITION - 1);
    TEMP.VALUE(POSITION .. (POSITION + ITEM.LEN - 1)) :=
      ITEM.VALUE(FIRST .. ITEM.LEN);
    TEMP.VALUE((POSITION + ITEM.LEN) .. TEMP.LEN) :=
      TARGET.VALUE(POSITION .. TARGET.LEN);

    return(TEMP);
  end INSERT;

function INSERT(TARGET: VSTRING; ITEM: STRING;
                POSITION : STRINDEX := FIRST) return VSTRING is
  begin -- INSERT
    return INSERT(TARGET, VSTR(ITEM), POSITION);
  end INSERT;

function INSERT(TARGET: VSTRING; ITEM: CHARACTER;
                POSITION : STRINDEX := FIRST) return VSTRING is
  begin -- INSERT
```

```
    return INSERT(TARGET, VSTR(ITEM), POSITION);
  end INSERT;


function APPEND(TARGET: VSTRING; ITEM: VSTRING; POSITION : STRINDEX)
                return VSTRING is

  TEMP : VSTRING;
  POS : STRINDEX := POSITION;

  begin -- APPEND
    if POSITION not in FIRST .. TARGET.LEN
      then raise CONSTRAINT_ERROR;
     end if;

    if TARGET.LEN + ITEM.LEN > LAST
      then raise CONSTRAINT_ERROR;
      else TEMP.LEN := TARGET.LEN + ITEM.LEN;
     end if;

    TEMP.VALUE(FIRST .. POS) := TARGET.VALUE(FIRST .. POS);
    TEMP.VALUE(POS + 1 .. (POS + ITEM.LEN)) := ITEM.VALUE(FIRST .. ITEM.LEN);
    TEMP.VALUE((POS + ITEM.LEN + 1) .. TEMP.LEN) :=
      TARGET.VALUE(POS + 1 .. TARGET.LEN);

    return(TEMP);
  end APPEND;

function APPEND(TARGET: VSTRING; ITEM: STRING; POSITION : STRINDEX)
                return VSTRING is
  begin -- APPEND
    return APPEND(TARGET, VSTR(ITEM), POSITION);
  end APPEND;

function APPEND(TARGET: VSTRING; ITEM: CHARACTER; POSITION : STRINDEX)
                return VSTRING is
  begin -- APPEND
    return APPEND(TARGET, VSTR(ITEM), POSITION);
  end APPEND;


function APPEND(TARGET: VSTRING; ITEM: VSTRING) return VSTRING is
  begin -- APPEND
    return(APPEND(TARGET, ITEM, TARGET.LEN));
  end APPEND;

function APPEND(TARGET: VSTRING; ITEM: STRING) return VSTRING is
  begin -- APPEND
    return(APPEND(TARGET, VSTR(ITEM), TARGET.LEN));
  end APPEND;
```

```
function APPEND(TARGET: VSTRING; ITEM: CHARACTER) return VSTRING is
  begin -- APPEND
    return(APPEND(TARGET, VSTR(ITEM), TARGET.LEN));
  end APPEND;


function REPLACE(TARGET: VSTRING; ITEM: VSTRING;
                 POSITION : STRINDEX := FIRST) return VSTRING is

  TEMP : VSTRING;

  begin -- REPLACE
    if POSITION not in FIRST .. TARGET.LEN
      then raise CONSTRAINT_ERROR;
     end if;

    if POSITION + ITEM.LEN - 1 <= TARGET.LEN
      then TEMP.LEN := TARGET.LEN;
      elsif POSITION + ITEM.LEN - 1 > LAST
        then raise CONSTRAINT_ERROR;
        else TEMP.LEN := POSITION + ITEM.LEN - 1;
     end if;

    TEMP.VALUE(FIRST .. POSITION - 1) := TARGET.VALUE(FIRST .. POSITION - 1);
    TEMP.VALUE(POSITION .. (POSITION + ITEM.LEN - 1)) :=
      ITEM.VALUE(FIRST .. ITEM.LEN);
    TEMP.VALUE((POSITION + ITEM.LEN) .. TEMP.LEN) :=
      TARGET.VALUE((POSITION + ITEM.LEN) .. TARGET.LEN);

    return(TEMP);
  end REPLACE;

function REPLACE(TARGET: VSTRING; ITEM: STRING;
                 POSITION : STRINDEX := FIRST) return VSTRING is
  begin -- REPLACE
    return REPLACE(TARGET, VSTR(ITEM), POSITION);
  end REPLACE;

function REPLACE(TARGET: VSTRING; ITEM: CHARACTER;
                 POSITION : STRINDEX := FIRST) return VSTRING is
  begin -- REPLACE
    return REPLACE(TARGET, VSTR(ITEM), POSITION);
  end REPLACE;


function "&"(LEFT:VSTRING; RIGHT : VSTRING) return VSTRING is

  TEMP : VSTRING;

  begin -- "&"
    if LEFT.LEN + RIGHT.LEN > LAST
```

```
      then raise CONSTRAINT_ERROR;
      else TEMP.LEN := LEFT.LEN + RIGHT.LEN;
     end if;

    TEMP.VALUE(FIRST .. TEMP.LEN) := LEFT.VALUE(FIRST .. LEFT.LEN) &
      RIGHT.VALUE(FIRST .. RIGHT.LEN);
    return(TEMP);
  end "&";

function "&"(LEFT:VSTRING; RIGHT : STRING) return VSTRING is
  begin -- "&"
    return LEFT & VSTR(RIGHT);
  end "&";

function "&"(LEFT:VSTRING; RIGHT : CHARACTER) return VSTRING is
  begin -- "&"
    return LEFT & VSTR(RIGHT);
  end "&";

function "&"(LEFT : STRING; RIGHT : VSTRING) return VSTRING is
  begin -- "&"
    return VSTR(LEFT) & RIGHT;
  end "&";

function "&"(LEFT : CHARACTER; RIGHT : VSTRING) return VSTRING is
  begin -- "&"
    return VSTR(LEFT) & RIGHT;
  end "&";


Function INDEX(WHOLE : VSTRING; PART : VSTRING; OCCURRENCE : NATURAL := 1)
               return STRINDEX is

  NOT_FOUND : constant NATURAL := 0;
  INDEX : NATURAL := FIRST;
  COUNT : NATURAL := 0;

  begin -- INDEX
    if PART = NUL then return(NOT_FOUND); -- by definition
      end if;

    while INDEX + PART.LEN - 1 <= WHOLE.LEN and then COUNT < OCCURRENCE loop
      if WHOLE.VALUE(INDEX .. PART.LEN + INDEX - 1) =
         PART.VALUE(1 .. PART.LEN)
        then COUNT := COUNT + 1;
       end if;
      INDEX := INDEX + 1;
     end loop;

    if COUNT = OCCURRENCE
      then return(INDEX - 1);
      else return(NOT_FOUND);
     end if;
  end INDEX;

Function INDEX(WHOLE : VSTRING; PART : STRING; OCCURRENCE : NATURAL := 1)
               return STRINDEX is

  begin -- Index
    return(Index(WHOLE, VSTR(PART), OCCURRENCE));
  end INDEX;


Function INDEX(WHOLE : VSTRING; PART : CHARACTER; OCCURRENCE : NATURAL := 1)
               return STRINDEX is

  begin -- Index
    return(Index(WHOLE, VSTR(PART), OCCURRENCE));
  end INDEX;


function RINDEX(WHOLE: VSTRING; PART:VSTRING; OCCURRENCE:NATURAL := 1)
               return STRINDEX is

  NOT_FOUND : constant NATURAL := 0;
  INDEX : INTEGER := WHOLE.LEN - (PART.LEN -1);
  COUNT : NATURAL := 0;

  begin -- RINDEX
    if PART = NUL then return(NOT_FOUND); -- by definition
      end if;

    while INDEX >= FIRST and then COUNT < OCCURRENCE loop
      if WHOLE.VALUE(INDEX .. PART.LEN + INDEX - 1) =
         PART.VALUE(1 .. PART.LEN)
        then COUNT := COUNT + 1;
       end if;
      INDEX := INDEX - 1;
     end loop;

    if COUNT = OCCURRENCE
      then
        if COUNT > 0
          then return(INDEX + 1);
          else return(NOT_FOUND);
         end if;
      else return(NOT_FOUND);
     end if;
  end RINDEX;

Function RINDEX(WHOLE : VSTRING; PART : STRING; OCCURRENCE : NATURAL := 1)
               return STRINDEX is
  begin -- Rindex
    return(RINDEX(WHOLE, VSTR(PART), OCCURRENCE));
  end RINDEX;


Function RINDEX(WHOLE : VSTRING; PART : CHARACTER; OCCURRENCE : NATURAL := 1)
               return STRINDEX is

  begin -- Rindex
    return(RINDEX(WHOLE, VSTR(PART), OCCURRENCE));
  end RINDEX;


function VSTR(FROM : CHARACTER) return VSTRING is

  TEMP : VSTRING;

  begin -- VSTR
    if LAST < 1
      then raise CONSTRAINT_ERROR;
      else TEMP.LEN := 1;
     end if;

    TEMP.VALUE(FIRST) := FROM;
    return(TEMP);
  end VSTR;


function VSTR(FROM : STRING) return VSTRING is

  TEMP : VSTRING;

  begin -- VSTR
    if FROM'LENGTH > LAST
      then raise CONSTRAINT_ERROR;
      else TEMP.LEN := FROM'LENGTH;
     end if;

    TEMP.VALUE(FIRST .. FROM'LENGTH) := FROM;
    return(TEMP);
  end VSTR;

Function "+" (FROM : STRING) return VSTRING is
  begin -- "+"
    return(VSTR(FROM));
  end "+";

Function "+" (FROM : CHARACTER) return VSTRING is
  begin
   return(VSTR(FROM));
```

```
    end "+";


  function CONVERT(X : FROM) return TO is

    begin -- CONVERT
      return(VSTR(STR(X)));
    end CONVERT;
end VSTRINGS;
```

```
--===========================================================================
-- FILES - "files.a" has the global data type declerations used by all the
--         other packages.
--===========================================================================
with VSTRINGS;
with SEQUENCES;
with GRAPHS;

package FILES is

  package VARSTRING is new VSTRINGS(80);
  use VARSTRING;

  subtype OPERATOR_ID is VSTRING;
  subtype VALUE is NATURAL;
  subtype MET is VALUE;
  subtype MRT is VALUE;
  subtype MCP is VALUE;
  subtype PERIOD is VALUE;
  subtype WITHIN is VALUE;
  subtype STARTS is VALUE;
  subtype STOPS is VALUE;
  subtype LOWERS is VALUE;
  subtype UPPERS is VALUE;

  Exception_Operator : OPERATOR_ID;

  TEST_VERIFIED : BOOLEAN := TRUE;

  type OPERATOR is
    record
      THE_OPERATOR_ID   : OPERATOR_ID;
      THE_MET           : MET  := 0;
      THE_MRT           : MRT  := 0;
      THE_MCP           : MCP  := 0;
      THE_PERIOD        : PERIOD  := 0;
      THE_WITHIN        : WITHIN  := 0;
    end record;

  package DIGRAPH is new GRAPHS(OPERATOR);

  type SCHEDULE_INPUTS is
    record
      THE_OPERATOR      : OPERATOR_ID;
      THE_START         : STARTS  := 0;
      THE_STOP          : STOPS   := 0;
      THE_LOWER         : LOWERS  := 0;
      THE_UPPER         : UPPERS  := 0;
    end record;

  package SCHEDULE_INPUTS_LIST is new SEQUENCES(SCHEDULE_INPUTS);
  type OP_INFO is
     record
       NODE         : OPERATOR;
       SUCCESSORS   : DIGRAPH.V_LISTS.LIST;
       PREDICESSORS : DIGRAPH.V_LISTS.LIST;
     end record;

   package OP_INFO_LIST is new SEQUENCES(OP_INFO);

 end FILES;
```

```
--=============================================================================
-- FILE_PROCESSOR - "fp_s.a, fp_b.a"; includes the procedures which are used to
--                  validate the information in the 'atomic.info' file and
--                  costruct the Graph Structure.
-- ============================================================================
with FILES; use FILES;
package FILE_PROCESSOR is

  procedure SEPARATE_DATA (THE_GRAPH : in out DIGRAPH.GRAPH);

  procedure VALIDATE_DATA (THE_GRAPH : in out DIGRAPH.GRAPH);

  CRIT_OP_LACKS_MET                   : exception;
  MET_NOT_LESS_THAN_PERIOD            : exception;
  MET_NOT_LESS_THAN_MRT               : exception;
  MCP_NOT_LESS_THAN_MRT               : exception;
  MCP_LESS_THAN_MET                   : exception;
  MET_IS_GREATER_THAN_FINISH_WITHIN : exception;
  SPORADIC_OP_LACKS_MCP               : exception;
  SPORADIC_OP_LACKS_MRT               : exception;
  PERIOD_LESS_THAN_FINISH_WITHIN     : exception;

end FILE_PROCESSOR;
--------------------------------
with TEXT_IO;
with FILES; use FILES;

package body FILE_PROCESSOR  is

  procedure SEPARATE_DATA (THE_GRAPH : in out DIGRAPH.GRAPH) is

    -- This procedure reads the output file which has the link information with
    -- the Atomic operators and depending upon the keywords that are declared
    -- as constants separates the information in the file and stores it in the
    -- graph data structure, where GRAPH has the operator and link information
    -- in it.

    package VALUE_IO is new TEXT_IO.INTEGER_IO(VALUE);

    MET     : constant VARSTRING.VSTRING := VARSTRING.VSTR("MET");
    MRT     : constant VARSTRING.VSTRING := VARSTRING.VSTR("MRT");
    MCP     : constant VARSTRING.VSTRING := VARSTRING.VSTR("MCP");
    PERIOD  : constant VARSTRING.VSTRING := VARSTRING.VSTR("PERIOD");
    WITHIN  : constant VARSTRING.VSTRING := VARSTRING.VSTR("WITHIN");
    LINK    : constant VARSTRING.VSTRING := VARSTRING.VSTR("LINK");
    ATOMIC  : constant VARSTRING.VSTRING := VARSTRING.VSTR("ATOMIC");
    EMPTY   : constant VARSTRING.VSTRING := VARSTRING.VSTR("EMPTY");

    Current_Value     : VALUE;
    New_Stream        : DIGRAPH.DATA_STREAM;
    New_Word          : VARSTRING.VSTRING;
  Cur_Opt           : OPERATOR;
  Cur_Link          : DIGRAPH.LINK_DATA;

  NON_CRITS    : TEXT_IO.FILE_TYPE;
  AG_OUTFILE   : TEXT_IO.FILE_TYPE;
  INPUT        : TEXT_IO.FILE_MODE := TEXT_IO.IN_FILE;
  OUTPUT       : TEXT_IO.FILE_MODE := TEXT_IO.OUT_FILE;
  PRINT_EDGES  : DIGRAPH.E_LISTS.LIST;
  S1, S2, L1   : DIGRAPH.V_LISTS.LIST;
  ID1, ID2     : OPERATOR;
  START_NODE   : OPERATOR;
  END_NODE     : OPERATOR;

  procedure INITIALIZE_OPERATOR (OP : in out OPERATOR) is
  begin
    OP.THE_MET    := 0;
    OP.THE_MRT    := 0;
    OP.THE_MCP    := 0;
    OP.THE_PERIOD := 0;
    OP.THE_WITHIN := 0;
  end;

begin
  TEXT_IO.OPEN (AG_OUTFILE, INPUT, "atomic.info");
  TEXT_IO.CREATE(NON_CRITS, OUTPUT, "non_crits");
  VARSTRING.GET_LINE (AG_OUTFILE, New_Word);
  while not TEXT_IO.END_OF_FILE(AG_OUTFILE) loop

    if VARSTRING.EQUAL (New_Word,LINK) then                -- keyword "LINK"
      START_NODE.THE_OPERATOR_ID := EMPTY;
      END_NODE.THE_OPERATOR_ID := EMPTY;
      DIGRAPH.V_STRING.GET_LINE(AG_OUTFILE,New_Stream);
      Cur_Link.THE_DATA_STREAM := New_Stream;
      VARSTRING.GET_LINE(AG_OUTFILE, New_Word);
      L1 := THE_GRAPH.VERTICES;
      while DIGRAPH.V_LISTS.NON_EMPTY(L1) loop
        if VARSTRING.EQUAL(DIGRAPH.V_LISTS.VALUE(L1).THE_OPERATOR_ID,New_Word)
                                                                       then
           START_NODE := DIGRAPH.V_LISTS.VALUE(L1);
           exit;
         end if;
         DIGRAPH.V_LISTS.NEXT(L1);
      end loop;
      VALUE_IO.GET(AG_OUTFILE, Current_Value);
      TEXT_IO.SKIP_LINE(AG_OUTFILE);
      Cur_Link.THE_LINK_MET := Current_Value;
      VARSTRING.GET_LINE (AG_OUTFILE, New_Word);
      L1 := THE_GRAPH.VERTICES;
      while DIGRAPH.V_LISTS.NON_EMPTY(L1) loop
        if VARSTRING.EQUAL(DIGRAPH.V_LISTS.VALUE(L1).THE_OPERATOR_ID,New_Word)
                                                                       then
        END_NODE := DIGRAPH.V_LISTS.VALUE(L1);
        exit;
      end if;
      DIGRAPH.V_LISTS.NEXT(L1);
   end loop;
   -- when either starting node or ending node of a link is EXTERNAL,
   -- the link information will not be added to the graph. Assuming
   -- that all external data coming in is ready at start time.

   if VARSTRING.NOTEQUAL(START_NODE.THE_OPERATOR_ID,EMPTY) and
               VARSTRING.NOTEQUAL(END_NODE.THE_OPERATOR_ID,EMPTY) then
     DIGRAPH.V_LISTS.ADD(START_NODE, Cur_Link.THE_FIRST_OP_ID);
     DIGRAPH.V_LISTS.ADD(END_NODE, Cur_Link.THE_SECOND_OP_ID);
     DIGRAPH.ADD(Cur_Link, THE_GRAPH);
   end if;
   VARSTRING.GET_LINE ( AG_OUTFILE, New_Word);

 elsif VARSTRING.EQUAL (New_Word,ATOMIC) then         -- keyword "ATOMIC"
   VARSTRING.GET_LINE ( AG_OUTFILE, New_Word);
   Cur_Opt.THE_OPERATOR_ID := New_Word;
   VARSTRING.GET_LINE (AG_OUTFILE, New_Word);
   if (VARSTRING.EQUAL(New_Word, ATOMIC)) or
                             (VARSTRING.EQUAL(New_Word, LINK)) or
                                (TEXT_IO.END_OF_FILE(AG_OUTFILE)) then
      VARSTRING.PUT_LINE(NON_CRITS, Cur_Opt.THE_OPERATOR_ID);
   else
     while VARSTRING.NOTEQUAL (New_Word, ATOMIC) and
           VARSTRING.NOTEQUAL (New_Word, LINK)   and
           not TEXT_IO.END_OF_FILE(AG_OUTFILE) loop

     if VARSTRING.EQUAL (New_Word,MET) then          -- keyword "MET"
         VALUE_IO.GET(AG_OUTFILE,Current_Value);
         TEXT_IO.SKIP_LINE(AG_OUTFILE);
         Cur_Opt.THE_MET := Current_Value;

       elsif VARSTRING.EQUAL (New_Word,MRT) then      -- keyword "MRT"
         VALUE_IO.GET(AG_OUTFILE,Current_Value);
         TEXT_IO.SKIP_LINE(AG_OUTFILE);
         Cur_Opt.THE_MRT:= Current_Value;

     elsif VARSTRING.EQUAL (New_Word,MCP) then       -- keyword "MCP"
         VALUE_IO.GET(AG_OUTFILE,Current_Value);
         TEXT_IO.SKIP_LINE(AG_OUTFILE);
       Cur_Opt.THE_MCP := Current_Value;

       elsif VARSTRING.EQUAL (New_Word,PERIOD) then   -- keyword "PERIOD"
         VALUE_IO.GET(AG_OUTFILE,Current_Value);
         TEXT_IO.SKIP_LINE(AG_OUTFILE);
         Cur_Opt.THE_PERIOD := Current_Value;

     elsif VARSTRING.EQUAL (New_Word,WITHIN) then   -- keyword "WITHIN"
```

```
            VALUE_IO.GET(AG_OUTFILE,Current_Value);
            TEXT_IO.SKIP_LINE(AG_OUTFILE);
            Cur_Opt.THE_WITHIN := Current_Value;
          end if;

          VARSTRING.GET_LINE(AG_OUTFILE,New_Word);
        end loop;

        DIGRAPH.ADD(Cur_Opt, THE_GRAPH);
          INITIALIZE_OPERATOR(Cur_OPt);
        end if;
      end if;
    end loop;
end SEPARATE_DATA;

procedure VALIDATE_DATA (THE_GRAPH : in out DIGRAPH.GRAPH) is
  -- check the correctness of the operator and the link information before
  -- running the algorithms. If any check fails in this procedure, the
  -- program halts.

  TARGET : DIGRAPH.V_LISTS.LIST;
  package VAL_IO is new TEXT_IO.INTEGER_IO(VALUE);
begin
  TARGET := THE_GRAPH.VERTICES;
  while DIGRAPH.V_LISTS.NON_EMPTY(TARGET) loop

    -- ensure that there is no operator without an MET.
    if DIGRAPH.V_LISTS.VALUE(TARGET).THE_MET = 0 then
      Exception_Operator := DIGRAPH.V_LISTS.VALUE(TARGET).THE_OPERATOR_ID;
      raise CRIT_OP_LACKS_MET;
    end if;

    if DIGRAPH.V_LISTS.VALUE(TARGET).THE_PERIOD = 0 then
      -- Check to ensure that MCP has a value for sporadic operators
      if DIGRAPH.V_LISTS.VALUE(TARGET).THE_MCP = 0 then
         Exception_Operator := DIGRAPH.V_LISTS.VALUE(TARGET).THE_OPERATOR_ID;
        raise SPORADIC_OP_LACKS_MCP;
      elsif  DIGRAPH.V_LISTS.VALUE(TARGET).THE_MET >
                                   DIGRAPH.V_LISTS.VALUE(TARGET).THE_MCP then
        Exception_Operator := DIGRAPH.V_LISTS.VALUE(TARGET).THE_OPERATOR_ID;
        raise MCP_LESS_THAN_MET;
      end if;

      -- Check to ensure that MRT has a value for sporadic operators
      if DIGRAPH.V_LISTS.VALUE(TARGET).THE_MRT = 0 then
        Exception_Operator := DIGRAPH.V_LISTS.VALUE(TARGET).THE_OPERATOR_ID;
        raise SPORADIC_OP_LACKS_MRT;
      end if;

      -- Check to ensure that the MRT is greater than the MET.
      if DIGRAPH.V_LISTS.VALUE(TARGET).THE_MET >
```

```
                                               DIGRAPH.V_LISTS.VALUE(TARGET).THE_MRT then
          Exception_Operator := DIGRAPH.V_LISTS.VALUE(TARGET).THE_OPERATOR_ID;
          raise MET_NOT_LESS_THAN_MRT;
        end if;

        -- Guarantees that an operator can fire at least once
        -- before a response expected.
        if DIGRAPH.V_LISTS.VALUE(TARGET).THE_MCP >
                                               DIGRAPH.V_LISTS.VALUE(TARGET).THE_MRT then
          raise MCP_NOT_LESS_THAN_MRT;
        end if;

     else
        -- Check to ensure that the PERIOD is greater than the MET.
        if DIGRAPH.V_LISTS.VALUE(TARGET).THE_MET >
                                            DIGRAPH.V_LISTS.VALUE(TARGET).THE_PERIOD then
          Exception_Operator := DIGRAPH.V_LISTS.VALUE(TARGET).THE_OPERATOR_ID;
          raise MET_NOT_LESS_THAN_PERIOD;
        end if;

        -- Check to ensure that the FINISH_WITHIN is grater than the MET.
        if DIGRAPH.V_LISTS.VALUE(TARGET).THE_WITHIN /= 0 then
          if DIGRAPH.V_LISTS.VALUE(TARGET).THE_MET >
                                            DIGRAPH.V_LISTS.VALUE(TARGET).THE_WITHIN then
            Exception_Operator := DIGRAPH.V_LISTS.VALUE(TARGET).THE_OPERATOR_ID;
            raise MET_IS_GREATER_THAN_FINISH_WITHIN;
          elsif DIGRAPH.V_LISTS.VALUE(TARGET).THE_PERIOD <
                                            DIGRAPH.V_LISTS.VALUE(TARGET).THE_WITHIN then
            Exception_Operator := DIGRAPH.V_LISTS.VALUE(TARGET).THE_OPERATOR_ID;
            raise PERIOD_LESS_THAN_FINISH_WITHIN;
          end if;
        end if;

      end if;
      DIGRAPH.V_LISTS.NEXT(TARGET);
    end loop;
  end VALIDATE_DATA;

end FILE_PROCESSOR;
```

```
--==============================================================================
-- TOPOLOGICAL_SORTER - "t_sort_s.a, t_sort_b.a"; this package contains one
                        procedure that does a topological sort of a linked list
--==============================================================================
with FILES;use FILES;
package TOPOLOGICAL_SORTER is

  procedure TOPOLOGICAL_SORT (G: in DIGRAPH.GRAPH;
                              PRECEDENCE_LIST: in out DIGRAPH.V_LISTS.LIST);

end TOPOLOGICAL_SORTER;
----------------------------
with TEXT_IO;
with FILES; use FILES;

package body TOPOLOGICAL_SORTER is

  -- This package determines the precedence order in which operators must
  -- execute in the final schedule.  This information is determined
  -- from the graph.

  procedure TOPOLOGICAL_SORT (G: in DIGRAPH.GRAPH;
                PRECEDENCE_LIST: in out DIGRAPH.V_LISTS.LIST) is

    -- This procedure determines which operators in the graph must
    -- be executed before another.

  Q : DIGRAPH.V_LISTS.LIST;

  begin

    DIGRAPH.T_SORT(G,PRECEDENCE_LIST);
    Q := PRECEDENCE_LIST;

  end TOPOLOGICAL_SORT;

end TOPOLOGICAL_SORTER;
```

```
--=========================================================================
-- HARMONIC_BLOCK_BUILDER - "hbb_s.a, hbb_b.a"; this package determines the
--                          periodic equivalents of the sporadic operators,
--                          and the length for the harmonic block.
--=========================================================================
with FILES; use FILES;
package HARMONIC_BLOCK_BUILDER is

  procedure CALC_PERIODIC_EQUIVALENTS (THE_GRAPH : in out DIGRAPH.GRAPH);

  procedure FIND_BASE_BLOCK (PRECEDENCE_LIST : in DIGRAPH.V_LISTS.LIST;
                             BASE_BLOCK      : out VALUE );

 procedure FIND_BLOCK_LENGTH (PRECEDENCE_LIST         : in DIGRAPH.V_LISTS.LIST;
                              HARMONIC_BLOCK_LENGTH   : out INTEGER );

  NO_BASE_BLOCK             : exception;
  NO_OPERATOR_IN_LIST       : exception;
  MET_NOT_LESS_THAN_PERIOD  : exception;

end HARMONIC_BLOCK_BUILDER;
---------------------------
with TEXT_IO;
with FILES; use FILES;
package body HARMONIC_BLOCK_BUILDER is

  procedure CALC_PERIODIC_EQUIVALENTS (THE_GRAPH : in out DIGRAPH.GRAPH) is

    V : DIGRAPH.V_LISTS.LIST := THE_GRAPH.VERTICES;
    E : DIGRAPH.E_LISTS.LIST := THE_GRAPH.LINKS;
    OPT : OPERATOR;
    NEW_P : VALUE := 0;
package val_io is new TEXT_IO.INTEGER_IO(value);

    procedure VERIFY_1 (O : in OPERATOR) is

    -- Check to ensure that MRT has a value for sporadic operators
    begin
      if O.THE_MET >= O.THE_PERIOD then
        Exception_Operator := O.THE_OPERATOR_ID;
        raise MET_NOT_LESS_THAN_PERIOD;
      end if;
    end VERIFY_1;

    procedure CALCULATE_NEW_PERIOD (O: in OPERATOR; NEW_PERIOD: in out VALUE) is
      DIFFERENCE : VALUE;
package VALUE_IO is new TEXT_IO.INTEGER_IO(VALUE);
    begin
      DIFFERENCE := O.THE_MRT - O.THE_MET;
      if DIFFERENCE < O.THE_MCP then
        NEW_PERIOD := DIFFERENCE;
      else
        NEW_PERIOD := O.THE_MCP;
      end if;
    end CALCULATE_NEW_PERIOD;

    procedure MODIFY_LINK_INFO (EDGES  : in out DIGRAPH.E_LISTS.LIST;
                                TARGET : in OPERATOR) is
      P          : DIGRAPH.E_LISTS.LIST := EDGES;
      START_NODE : DIGRAPH.V_LISTS.LIST;
      END_NODE   : DIGRAPH.V_LISTS.LIST;
      V1, V2     : OPERATOR;
    begin
      while DIGRAPH.E_LISTS.NON_EMPTY(P) loop
        START_NODE := DIGRAPH.E_LISTS.VALUE(P).THE_FIRST_OP_ID;
        END_NODE   := DIGRAPH.E_LISTS.VALUE(P).THE_SECOND_OP_ID;
        V1         := DIGRAPH.V_LISTS.VALUE(START_NODE);
        V2         := DIGRAPH.V_LISTS.VALUE(END_NODE);
        if VARSTRING.EQUAL(V1.THE_OPERATOR_ID, TARGET.THE_OPERATOR_ID) then
          START_NODE.ELEMENT.THE_PERIOD := TARGET.THE_PERIOD;
        elsif VARSTRING.EQUAL(V2.THE_OPERATOR_ID, TARGET.THE_OPERATOR_ID) then
          END_NODE.ELEMENT.THE_PERIOD := TARGET.THE_PERIOD;
        end if;
        DIGRAPH.E_LISTS.NEXT(P);
      end loop;
    end MODIFY_LINK_INFO;

  begin -- main CALC_PERIODIC_EQUIVALENTS
    while DIGRAPH.V_LISTS.NON_EMPTY(V) loop
      OPT := DIGRAPH.V_LISTS.VALUE(V);
      if OPT.THE_PERIOD = 0 then
        CALCULATE_NEW_PERIOD(OPT, NEW_P);
        OPT.THE_PERIOD := NEW_P;
        VERIFY_1(OPT);
        MODIFY_LINK_INFO(E, OPT);
        V.element.the_period := new_p;
        E := THE_GRAPH.LINKS;
      end if;
      DIGRAPH.V_LISTS.NEXT(V);
    end loop;
  end CALC_PERIODIC_EQUIVALENTS;

  procedure FIND_BASE_BLOCK (PRECEDENCE_LIST : in DIGRAPH.V_LISTS.LIST;
                             BASE_BLOCK      : out VALUE ) is

    P_LIST : DIGRAPH.V_LISTS.LIST := PRECEDENCE_LIST;
    DIVISOR : VALUE;
    ALTERNATE_SEQUENCE  : DIGRAPH.V_LISTS.LIST;
    BASE_BLOCK_SEQUENCE : DIGRAPH.V_LISTS.LIST;

    function FIND_MINIMUM_PERIOD (P_LIST : in DIGRAPH.V_LISTS.LIST)
                                                        return VALUE is
    V : DIGRAPH.V_LISTS.LIST := P_LIST;
    MIN_PERIOD : VALUE := 0;

  begin
    if DIGRAPH.V_LISTS.NON_EMPTY(V) then
      MIN_PERIOD := DIGRAPH.V_LISTS.VALUE(V).THE_PERIOD;
      DIGRAPH.V_LISTS.NEXT(V);
      while DIGRAPH.V_LISTS.NON_EMPTY(V) loop
        if DIGRAPH.V_LISTS.VALUE(V).THE_PERIOD < MIN_PERIOD then
          MIN_PERIOD := DIGRAPH.V_LISTS.VALUE(V).THE_PERIOD;
        end if;
        DIGRAPH.V_LISTS.NEXT(V);
      end loop;
      return MIN_PERIOD;
    else
      raise NO_OPERATOR_IN_LIST;
    end if;
  end FIND_MINIMUM_PERIOD;

  function MODE_DIVIDE (THE_PERIOD : in VALUE) return VALUE is
  begin
    return (THE_PERIOD mod DIVISOR);
  end MODE_DIVIDE;

  procedure INITIAL_PASS (P_LIST : in out DIGRAPH.V_LISTS.LIST;
                          BASE_BLOCK_SEQUENCE : in out DIGRAPH.V_LISTS.LIST;
                          ALTERNATE_SEQUENCE  : in out DIGRAPH.V_LISTS.LIST) is

    ORIG_SEQUENCE : DIGRAPH.V_LISTS.LIST := P_LIST;
    OP_FROM_ORG_SEQ : OPERATOR;
    REMAINDER : VALUE;
    THE_PERIOD : VALUE;
  begin
    while DIGRAPH.V_LISTS.NON_EMPTY(ORIG_SEQUENCE) loop
      THE_PERIOD := DIGRAPH.V_LISTS.VALUE(ORIG_SEQUENCE).THE_PERIOD;
      REMAINDER := MODE_DIVIDE (THE_PERIOD);
      OP_FROM_ORG_SEQ := DIGRAPH.V_LISTS.VALUE(ORIG_SEQUENCE);
      if REMAINDER = 0 then
        DIGRAPH.V_LISTS.ADD (OP_FROM_ORG_SEQ, BASE_BLOCK_SEQUENCE);
      else
        DIGRAPH.V_LISTS.ADD (OP_FROM_ORG_SEQ, ALTERNATE_SEQUENCE);
      end if;
      DIGRAPH.V_LISTS.NEXT(ORIG_SEQUENCE);
    end loop;
  end INITIAL_PASS;

begin -- main FIND_BASE_BLOCK
  DIVISOR := FIND_MINIMUM_PERIOD(P_LIST);
  INITIAL_PASS(P_LIST, BASE_BLOCK_SEQUENCE, ALTERNATE_SEQUENCE);
  while DIGRAPH.V_LISTS.NON_EMPTY(ALTERNATE_SEQUENCE) loop
    if DIVISOR = 1 then
```

```
      raise NO_BASE_BLOCK;
      -- exit and terminate the Static Scheduler
    else
      DIVISOR := DIVISOR - 1;
      ALTERNATE_SEQUENCE  := null;
      BASE_BLOCK_SEQUENCE := null;
      INITIAL_PASS(P_LIST, BASE_BLOCK_SEQUENCE, ALTERNATE_SEQUENCE);
    end if;
  end loop;
  BASE_BLOCK := DIVISOR;
end FIND_BASE_BLOCK;

procedure FIND_BLOCK_LENGTH (PRECEDENCE_LIST       : in DIGRAPH.V_LISTS.LIST;
                             HARMONIC_BLOCK_LENGTH : out INTEGER ) is

  ORIG_SEQUENCE : DIGRAPH.V_LISTS.LIST := PRECEDENCE_LIST;
  NUMBER1    : VALUE;
  NUMBER2    : VALUE;
  LCM        : VALUE;
  GCD        : VALUE;
  TARGET_NO  : VALUE;

  function FIND_GCD (NUMBER1 : in VALUE; NUMBER2 : in VALUE) return VALUE is
    NEW_GCD    : VALUE;
  begin
    while GCD /= 0 loop
      if (NUMBER1 mod GCD = 0) and (NUMBER2 mod GCD = 0) then
        NEW_GCD := GCD;
        return NEW_GCD;
      else
        GCD := GCD - 1;
      end if;
    end loop;
  end FIND_GCD;

  function FIND_LCM (NUMBER1, NUMBER2 : VALUE) return VALUE is
  begin
    return(NUMBER1 * NUMBER2) / GCD;
   end FIND_LCM;

begin -- main FIND_BLOCK_LENGTH
  if DIGRAPH.V_LISTS.NON_EMPTY(ORIG_SEQUENCE) then
    NUMBER1 := DIGRAPH.V_LISTS.VALUE(ORIG_SEQUENCE).THE_PERIOD;
    DIGRAPH.V_LISTS.NEXT(ORIG_SEQUENCE);
    while DIGRAPH.V_LISTS.NON_EMPTY(ORIG_SEQUENCE) loop
      NUMBER2 := DIGRAPH.V_LISTS.VALUE(ORIG_SEQUENCE).THE_PERIOD;
      if NUMBER2 > NUMBER1 then
        GCD := NUMBER1;
        TARGET_NO := NUMBER2;
      else
        GCD := NUMBER2;
```

```
          TARGET_NO := NUMBER1;
        end if;
        GCD := FIND_GCD(GCD, TARGET_NO);
        LCM := FIND_LCM(NUMBER1, NUMBER2);
        NUMBER1 := LCM;
        DIGRAPH.V_LISTS.NEXT(ORIG_SEQUENCE);
      end loop;
      HARMONIC_BLOCK_LENGTH := LCM;
    else
      raise NO_OPERATOR_IN_LIST;
    end if;
  end FIND_BLOCK_LENGTH;

end HARMONIC_BLOCK_BUILDER;
```

```
--==========================================================================
-- OPERATOR_SCHEDULER - "scheduler_s.a, scheduler_b.a"; contains all the
--                      scheduling algorithms implemented. It creates a static
--                      schedule into the 'ss.a' file, if possible.
--==========================================================================
with FILES; use FILES;
package OPERATOR_SCHEDULER is

  procedure TEST_DATA (INPUT_LIST           : in DIGRAPH.V_LISTS.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER);

  procedure SCHEDULE_INITIAL_SET (PRECEDENCE_LIST : in DIGRAPH.V_LISTS.LIST;
                       THE_SCHEDULE_INPUTS   : in out SCHEDULE_INPUTS_LIST.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER;
                       STOP_TIME             : in out INTEGER);

  procedure SCHEDULE_REST_OF_BLOCK(PRECEDENCE_LIST : in DIGRAPH.V_LISTS.LIST;
                       THE_SCHEDULE_INPUTS   : in out SCHEDULE_INPUTS_LIST.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER;
                       STOP_TIME             : in INTEGER);

  procedure SCHEDULE_WITH_EARLIEST_START (THE_GRAPH : in DIGRAPH.GRAPH;
                       AGENDA                : in out SCHEDULE_INPUTS_LIST.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER);

  procedure SCHEDULE_WITH_EARLIEST_DEADLINE (THE_GRAPH : in DIGRAPH.GRAPH;
                       AGENDA                : in out SCHEDULE_INPUTS_LIST.LIST;
                       HARMONIC_BLOCK_LENGTH : in INTEGER);

  procedure CREATE_STATIC_SCHEDULE (THE_GRAPH     : in DIGRAPH.GRAPH;
                           THE_SCHEDULE_INPUTS   : in SCHEDULE_INPUTS_LIST.LIST;
                           HARMONIC_BLOCK_LENGTH : in INTEGER);

  MISSED_DEADLINE  : exception;
  OVER_TIME        : exception;
  MISSED_OPERATOR  : exception;

end OPERATOR_SCHEDULER;
------------------------------
with FILES; use FILES;
with TEXT_IO;
package body OPERATOR_SCHEDULER is

  procedure TEST_DATA  (INPUT_LIST            : in DIGRAPH.V_LISTS.LIST;
                        HARMONIC_BLOCK_LENGTH : in INTEGER) is

    procedure CALC_TOTAL_TIME (INPUT_LIST            : in DIGRAPH.V_LISTS.LIST;
                               HARMONIC_BLOCK_LENGTH : in INTEGER) is
      V : DIGRAPH.V_LISTS.LIST := INPUT_LIST;
      TIMES       : FLOAT := 0.0;
      OP_TIME     : FLOAT := 0.0;
    TOTAL_TIME : FLOAT := 0.0;
    PER        : OPERATOR;
    BAD_TOTAL_TIME : exception;

    function CALC_NO_OF_PERIODS (HARMONIC_BLOCK_LENGTH : in INTEGER;
                                 THE_PERIOD : in INTEGER) return FLOAT is
    begin
      return FLOAT(HARMONIC_BLOCK_LENGTH) / FLOAT(THE_PERIOD);
    end CALC_NO_OF_PERIODS;

    function MULTIPLY_BY_MET (TIMES   : in FLOAT;
                              THE_MET : in VALUE) return FLOAT is
    begin
      return TIMES * FLOAT(THE_MET);
    end MULTIPLY_BY_MET;

    function ADD_TO_SUM (OP_TIME : in FLOAT) return FLOAT is
    begin
      return TOTAL_TIME + OP_TIME;
    end ADD_TO_SUM;

  begin --main CALC_TOTAL_TIME
    while DIGRAPH.V_LISTS.NON_EMPTY(V) loop
     PER := DIGRAPH.V_LISTS.VALUE(V);
     TIMES:= CALC_NO_OF_PERIODS (HARMONIC_BLOCK_LENGTH , PER.THE_PERIOD);
     OP_TIME := MULTIPLY_BY_MET (TIMES, DIGRAPH.V_LISTS.VALUE(V).THE_MET);
     TOTAL_TIME := ADD_TO_SUM (OP_TIME);
     if TOTAL_TIME > FLOAT(HARMONIC_BLOCK_LENGTH) then
       raise BAD_TOTAL_TIME;
     else
       DIGRAPH.V_LISTS.NEXT(V);
     end if;
    end loop;
    exception
      when BAD_TOTAL_TIME =>
           TEST_VERIFIED := FALSE;
           TEXT_IO.PUT("The total execution time of the operators exceeds ");
           TEXT_IO.PUT_LINE("the HARMONIC_BLOCK_LENGTH");
           TEXT_IO.NEW_LINE;
  end CALC_TOTAL_TIME;

  procedure CALC_HALF_PERIODS (INPUT_LIST : in DIGRAPH.V_LISTS.LIST) is

    V : DIGRAPH.V_LISTS.LIST := INPUT_LIST;
    HALF_PERIOD : FLOAT;
    FAIL_HALF_PERIOD : exception;

    function DIVIDE_PERIOD_BY_2 (THE_PERIOD : in VALUE) return FLOAT is
    begin
      return FLOAT(THE_PERIOD) / 2.0;
    end DIVIDE_PERIOD_BY_2;
```

```
begin --main CALC_HALF_PERIODS;
  while DIGRAPH.V_LISTS.NON_EMPTY(V) loop
    HALF_PERIOD := DIVIDE_PERIOD_BY_2(DIGRAPH.V_LISTS.VALUE(V).THE_PERIOD);
    if FLOAT(DIGRAPH.V_LISTS.VALUE(V).THE_MET) > HALF_PERIOD then
      Exception_Operator := DIGRAPH.V_LISTS.VALUE(V).THE_OPERATOR_ID;
      raise FAIL_HALF_PERIOD;
    else
      DIGRAPH.V_LISTS.NEXT(V);
    end if;
  end loop;
  exception
    when FAIL_HALF_PERIOD =>
         TEST_VERIFIED := FALSE;
         TEXT_IO.PUT ("The MET of Operator ");
         VARSTRING.PUT (Exception_Operator);
         TEXT_IO.PUT_LINE (" is greater than half of its period.");
end CALC_HALF_PERIODS;

procedure CALC_RATIO_SUM (INPUT_LIST : in DIGRAPH.V_LISTS.LIST) is
  V : DIGRAPH.V_LISTS.LIST := INPUT_LIST;
  RATIO : FLOAT;
  RATIO_SUM : FLOAT := 0.0;
  THE_MET : VALUE;
  THE_PERIOD : VALUE;
  RATIO_TOO_BIG : exception;

  function DIVIDE_MET_BY_PERIOD (THE_MET : in VALUE;
                                 THE_PERIOD : in VALUE) return FLOAT is
  begin
    return FLOAT(THE_MET) / FLOAT(THE_PERIOD);
  end DIVIDE_MET_BY_PERIOD;

  function ADD_TO_TIME (RATIO : in FLOAT) return FLOAT is
  begin
    return RATIO_SUM + RATIO;
  end ADD_TO_TIME;

begin --main CALC_RATIO_SUM
  while DIGRAPH.V_LISTS.NON_EMPTY(V)  loop
    THE_MET :=  DIGRAPH.V_LISTS.VALUE(V).THE_MET;
    THE_PERIOD := DIGRAPH.V_LISTS.VALUE(V).THE_PERIOD;
    RATIO := DIVIDE_MET_BY_PERIOD(THE_MET,THE_PERIOD);
    RATIO_SUM := ADD_TO_TIME(RATIO);
    DIGRAPH.V_LISTS.NEXT(V);
  end loop;
  if RATIO_SUM > 0.5 then
    raise RATIO_TOO_BIG;
  end if;
  exception
    when RATIO_TOO_BIG =>
         TEST_VERIFIED := FALSE;
```

```
            TEXT_IO.PUT ("The total MET/PERIOD ratio sum of operators is ");
            TEXT_IO.PUT_LINE ("greater than 0.5");
    end CALC_RATIO_SUM;

  begin --main TEST_DATA
    CALC_TOTAL_TIME(INPUT_LIST, HARMONIC_BLOCK_LENGTH);
    CALC_HALF_PERIODS(INPUT_LIST);
    CALC_RATIO_SUM(INPUT_LIST);
  end TEST_DATA;
------------------------------------------------
  procedure VERIFY_TIME_LEFT (HARMONIC_BLOCK_LENGTH : in INTEGER;
                              STOP_TIME : in INTEGER) is
  begin
    if STOP_TIME > HARMONIC_BLOCK_LENGTH then
      raise OVER_TIME;
      --exit and terminate the Static Scheduler
    end if;
  end VERIFY_TIME_LEFT;
------------------------------------------------
  procedure CREATE_INTERVAL (THE_OPERATOR : in OPERATOR;
                             INPUT        : in out SCHEDULE_INPUTS;
                             OLD_LOWER    : in VALUE) is
    LOWER_BOUND : VALUE;

    function CALC_LOWER_BOUND return VALUE is
    begin
      -- since CREATE_INTERVAL function is used in both SCHEDULE_INITIAL_SET and
      -- SCHEDULE_REST_OF_BLOCK (OLD_LOWER /= 0) check is needed.In case of the
      -- operator is scheduled somewhere in its interval and (OLD_LOWER /= 0),
      -- this check guarantees that the periods will be consistent.
      if (OLD_LOWER /= 0) and (OLD_LOWER < INPUT.THE_START) then
        LOWER_BOUND := OLD_LOWER + THE_OPERATOR.THE_PERIOD;
      else
        LOWER_BOUND := INPUT.THE_START + THE_OPERATOR.THE_PERIOD;
      end if;
      return LOWER_BOUND;
    end CALC_LOWER_BOUND;

    function CALC_UPPER_BOUND return VALUE is
    begin
      if THE_OPERATOR.THE_WITHIN = 0 then
        return LOWER_BOUND + THE_OPERATOR.THE_PERIOD - THE_OPERATOR.THE_MET;
      -- if the operator has a WITHIN constraint, the upper bound of the
      -- interval is reduced.
      else
        return LOWER_BOUND + THE_OPERATOR.THE_WITHIN - THE_OPERATOR.THE_MET;
      end if;
    end CALC_UPPER_BOUND;
  begin --main CREATE_INTERVAL
    INPUT.THE_LOWER := CALC_LOWER_BOUND;
    INPUT.THE_UPPER := CALC_UPPER_BOUND;
  end CREATE_INTERVAL;
------------------------------------------------
  procedure SCHEDULE_INITIAL_SET (PRECEDENCE_LIST : in DIGRAPH.V_LISTS.LIST;
                    THE_SCHEDULE_INPUTS   : in out SCHEDULE_INPUTS_LIST.LIST;
                    HARMONIC_BLOCK_LENGTH : in INTEGER;
                    STOP_TIME             : in out INTEGER) is

    V : DIGRAPH.V_LISTS.LIST := PRECEDENCE_LIST;
    START_TIME : INTEGER := 0;
    NEW_INPUT  : SCHEDULE_INPUTS;
    OLD_LOWER  : VALUE :=0;
    package INTEGERIO is new TEXT_IO.INTEGER_IO(INTEGER);
    use INTEGERIO;

  begin --SCEDULE_INITIAL_SET
    while  DIGRAPH.V_LISTS.NON_EMPTY(V) loop
      Exception_Operator := DIGRAPH.V_LISTS.VALUE(V).THE_OPERATOR_ID;
      NEW_INPUT.THE_OPERATOR := DIGRAPH.V_LISTS.VALUE(V).THE_OPERATOR_ID;
      NEW_INPUT.THE_START := START_TIME;
      STOP_TIME := START_TIME + DIGRAPH.V_LISTS.VALUE(V).THE_MET;
      VERIFY_TIME_LEFT(HARMONIC_BLOCK_LENGTH, STOP_TIME);
      NEW_INPUT.THE_STOP := STOP_TIME;
      START_TIME := STOP_TIME;
      -- for every operator in SCHEDULE_INITIAL_SET, OLD_LOWER is zero. So we
      -- always send zero value to CREATE_INTERVAL.
      CREATE_INTERVAL(DIGRAPH.V_LISTS.VALUE(V), NEW_INPUT, OLD_LOWER);
      SCHEDULE_INPUTS_LIST.ADD (NEW_INPUT, THE_SCHEDULE_INPUTS);
      DIGRAPH.V_LISTS.NEXT(V);
    end loop;
  end SCHEDULE_INITIAL_SET;
-------------------------------------------------
  procedure SCHEDULE_REST_OF_BLOCK(PRECEDENCE_LIST:in DIGRAPH.V_LISTS.LIST;
                    THE_SCHEDULE_INPUTS   : in out SCHEDULE_INPUTS_LIST.LIST;
                    HARMONIC_BLOCK_LENGTH : in INTEGER;
                    STOP_TIME             : in INTEGER) is

    V : DIGRAPH.V_LISTS.LIST := PRECEDENCE_LIST;
    TEMP : SCHEDULE_INPUTS_LIST.LIST := THE_SCHEDULE_INPUTS;
    V_LIST  : DIGRAPH.V_LISTS.LIST;
    P : SCHEDULE_INPUTS_LIST.LIST;
    S : SCHEDULE_INPUTS_LIST.LIST;
    START_TIME : INTEGER := 0;
    TIME_STOP  : INTEGER := STOP_TIME;
    NEW_INPUT  : SCHEDULE_INPUTS;
    OLD_LOWER  : VALUE;

package INTEGERIO is new TEXT_IO.INTEGER_IO(INTEGER);
use INTEGERIO;

  begin
    DIGRAPH.V_LISTS.DUPLICATE(PRECEDENCE_LIST, V_LIST);
    SCHEDULE_INPUTS_LIST.LIST_REVERSE(THE_SCHEDULE_INPUTS, P);

   loop
    while SCHEDULE_INPUTS_LIST.NON_EMPTY(P) loop
      if SCHEDULE_INPUTS_LIST.VALUE(P).THE_LOWER < HARMONIC_BLOCK_LENGTH then
      NEW_INPUT.THE_OPERATOR := DIGRAPH.V_LISTS.VALUE(V).THE_OPERATOR_ID;
        -- check if the operator can be scheduled in its interval
        if SCHEDULE_INPUTS_LIST.VALUE(P).THE_UPPER - TIME_STOP
                                            >= DIGRAPH.V_LISTS.VALUE(V).THE_MET then
          if SCHEDULE_INPUTS_LIST.VALUE(P).THE_LOWER >= TIME_STOP then
            START_TIME := SCHEDULE_INPUTS_LIST.VALUE(P).THE_LOWER;
          else
            START_TIME := TIME_STOP;
          end if;
        NEW_INPUT.THE_START := START_TIME;
        NEW_INPUT.THE_STOP := START_TIME + DIGRAPH.V_LISTS.VALUE(V).THE_MET;
          TIME_STOP := NEW_INPUT.THE_STOP;
          OLD_LOWER := SCHEDULE_INPUTS_LIST.VALUE(P).THE_LOWER;
        CREATE_INTERVAL(DIGRAPH.V_LISTS.VALUE(V), NEW_INPUT, OLD_LOWER);
        SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT, TEMP);
        SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT, S);
          Exception_Operator := DIGRAPH.V_LISTS.VALUE(V).THE_OPERATOR_ID;
        VERIFY_TIME_LEFT(HARMONIC_BLOCK_LENGTH, TIME_STOP);
          DIGRAPH.V_LISTS.NEXT(V);
        SCHEDULE_INPUTS_LIST.NEXT(P);
        -- if the operator can not be scheduled in its interval raise the
        -- exception
      else
          Exception_Operator := DIGRAPH.V_LISTS.VALUE(V).THE_OPERATOR_ID;
        raise MISSED_DEADLINE;
      end if;
      else
        DIGRAPH.V_LISTS.REMOVE
                     (DIGRAPH.V_LISTS.VALUE(V), V_LIST);
        DIGRAPH.V_LISTS.NEXT(V);
      SCHEDULE_INPUTS_LIST.NEXT(P);
      end if;
    end loop;
    if SCHEDULE_INPUTS_LIST.NON_EMPTY(S) then
      SCHEDULE_INPUTS_LIST.LIST_REVERSE(S, P);
      SCHEDULE_INPUTS_LIST.EMPTY(S);
      V := V_LIST;
    else
      exit;
    end if;
    end loop;
    SCHEDULE_INPUTS_LIST.LIST_REVERSE(TEMP, THE_SCHEDULE_INPUTS);

  end SCHEDULE_REST_OF_BLOCK;
-------------------------------------------------
  procedure BUILD_OP_INFO_LIST(THE_GRAPH         : in DIGRAPH.GRAPH;
                                THE_OP_INFO_LIST : in out OP_INFO_LIST.LIST) is
    -- this procedure finds each operator's successors and predicessors first
    -- and creates the OPERATOR_INFO_LIST.
    V : DIGRAPH.V_LISTS.LIST := THE_GRAPH.VERTICES;
    S : DIGRAPH.V_LISTS.LIST;
    P : DIGRAPH.V_LISTS.LIST;
    NEW_NODE : OP_INFO;

  begin
    while DIGRAPH.V_LISTS.NON_EMPTY(V) loop
      DIGRAPH.SCAN_CHILDREN(DIGRAPH.V_LISTS.VALUE(V), THE_GRAPH, S);
      DIGRAPH.SCAN_PARENTS(DIGRAPH.V_LISTS.VALUE(V), THE_GRAPH, P);
      NEW_NODE.NODE := DIGRAPH.V_LISTS.VALUE(V);
      NEW_NODE.SUCCESSORS := S;
      NEW_NODE.PREDICESSORS := P;
      OP_INFO_LIST.ADD(NEW_NODE, THE_OP_INFO_LIST);
      DIGRAPH.V_LISTS.NEXT(V);
    end loop;
  end BUILD_OP_INFO_LIST;
-------------------------------------------------
  procedure PROCESS_EST_END_NODE
                          (MAY_BE_AVAILABLE: in out SCHEDULE_INPUTS_LIST.LIST;
                           OPT               : in OPERATOR) is
    -- transfer the OPERATOR record into SCHEDULE_INFO record and adds that
    -- into the MAY_AVAILABLE_LIST for the Earliest Start Scheduling Algorithm.
    -- Initially all the values are zero.
    NEW_NODE : SCHEDULE_INPUTS;

  begin
    NEW_NODE.THE_OPERATOR := OPT.THE_OPERATOR_ID;
    SCHEDULE_INPUTS_LIST.ADD(NEW_NODE, MAY_BE_AVAILABLE);
  end PROCESS_EST_END_NODE;
-------------------------------------------------
  procedure PROCESS_EDL_END_NODE
                          (MAY_BE_AVAILABLE: in out SCHEDULE_INPUTS_LIST.LIST;
                           OPT               : in OPERATOR) is
    --transfer the OPERATOR record into SCHEDULE_INFO record and adds that
    --into the MAY_AVAILABLE_LIST for the Earliest Deadline Scheduling Algorithm
    --Initially all the values are zero.
    NEW_NODE : SCHEDULE_INPUTS;

  begin
    NEW_NODE.THE_OPERATOR := OPT.THE_OPERATOR_ID;
    NEW_NODE.THE_LOWER := 0;    -- we can omit this, because it's already zero.
    if OPT.THE_WITHIN /= 0 then
      NEW_NODE.THE_UPPER := OPT.THE_WITHIN;
    else
      NEW_NODE.THE_UPPER := OPT.THE_PERIOD;
    end if;
    SCHEDULE_INPUTS_LIST.ADD(NEW_NODE, MAY_BE_AVAILABLE);
  end PROCESS_EDL_END_NODE;
```

```
------------------------------------------------
  function FIND_OPERATOR(THE_OP_INFO_LIST : in OP_INFO_LIST.LIST;
                         ID               : in OPERATOR_ID)
                                                        return OP_INFO_LIST.LIST is
    -- finds the operator that we use currently to get the required information.
    TEMP : OP_INFO_LIST.LIST := THE_OP_INFO_LIST;

  -- assumed that it's guaranteed to find an operator.
  begin
    while OP_INFO_LIST.NON_EMPTY(TEMP) loop
      if VARSTRING.EQUAL(OP_INFO_LIST.VALUE(TEMP).NODE.THE_OPERATOR_ID, ID) then
        return TEMP ;
      end if;
      OP_INFO_LIST.NEXT(TEMP);
    end loop;
  end FIND_OPERATOR;
------------------------------------------------
  function CHECK_AGENDA(THE_NODE : in OP_INFO;
                        AGENDA   : in SCHEDULE_INPUTS_LIST.LIST)
                                                                 return BOOLEAN is
    -- checks the AGENDA list to see if all the predicessors of the operator are
    -- in there.
    P  : DIGRAPH.V_LISTS.LIST := THE_NODE.PREDICESSORS;
    A  : SCHEDULE_INPUTS_LIST.LIST := AGENDA;
    OK : BOOLEAN := FALSE;
  begin
    while DIGRAPH.V_LISTS.NON_EMPTY(P) loop
      while SCHEDULE_INPUTS_LIST.NON_EMPTY(A) loop
        if VARSTRING.EQUAL(DIGRAPH.V_LISTS.VALUE(P).THE_OPERATOR_ID,
                           SCHEDULE_INPUTS_LIST.VALUE(A).THE_OPERATOR) then
          OK := TRUE;
          exit;
        end if;
        SCHEDULE_INPUTS_LIST.NEXT(A);
      end loop;
      if OK then
        DIGRAPH.V_LISTS.NEXT(P);
        A := AGENDA;
        OK := FALSE;
      else
        -- if the pointer reached to the end of the AGENDA, it means the
        -- operator is not in AGENDA, if so return FALSE.
        return OK;
      end if;
    end loop;
    -- if the pointer reached to the end of the predicessor list, it means the
    -- operator is in AGENDA.
    OK := TRUE;
    return OK;
  end CHECK_AGENDA;
------------------------------------------------
```

```
procedure EST_INSERT (TARGET           : in SCHEDULE_INPUTS;
                      MAY_BE_AVAILABLE : in out SCHEDULE_INPUTS_LIST.LIST) is
  -- used to insert the operators into the MAY_BE_AVAILABLE list to schedule
  -- for the Earliest Start Scheduling Algorithm.
  PREV : SCHEDULE_INPUTS_LIST.LIST := null;
  T    : SCHEDULE_INPUTS_LIST.LIST := MAY_BE_AVAILABLE;

begin
  if NOT(SCHEDULE_INPUTS_LIST.NON_EMPTY(T)) then
    -- when MAY_BE_AVAILABLE list is empty, add the operator immediately.
    SCHEDULE_INPUTS_LIST.ADD(TARGET, MAY_BE_AVAILABLE);
  else

    -- in case the target operator's EST is smaller than the first operator's
    -- EST add the operator to the list immediately.
    if TARGET.THE_LOWER < SCHEDULE_INPUTS_LIST.VALUE(T).THE_LOWER then
      SCHEDULE_INPUTS_LIST.ADD(TARGET, MAY_BE_AVAILABLE);

    -- in case the operator with the same EST is in the list, do not insert,
    -- otherwise;  insert the operator in its order.
    elsif NOT(SCHEDULE_INPUTS_LIST.MEMBER(TARGET, MAY_BE_AVAILABLE)) then
      while SCHEDULE_INPUTS_LIST.NON_EMPTY(T) loop
        if TARGET.THE_LOWER > SCHEDULE_INPUTS_LIST.VALUE(T).THE_LOWER then
          PREV := T;
          SCHEDULE_INPUTS_LIST.NEXT(T);
        else
          exit;
        end if;
      end loop;
      SCHEDULE_INPUTS_LIST.ADD(TARGET, T);
      if SCHEDULE_INPUTS_LIST.NON_EMPTY(PREV) then
        PREV.NEXT := T;
      else
        MAY_BE_AVAILABLE := T;
      end if;
    end if;
  end if;
end EST_INSERT;
------------------------------------------------
procedure EDL_INSERT (TARGET           : in SCHEDULE_INPUTS;
                      MAY_BE_AVAILABLE : in out SCHEDULE_INPUTS_LIST.LIST) is

  -- used to insert the operators into the MAY_BE_AVAILABLE list to schedule
  -- for the Earliest Deadline Scheduling Algorithm.
  PREV : SCHEDULE_INPUTS_LIST.LIST := null;
  T    : SCHEDULE_INPUTS_LIST.LIST := MAY_BE_AVAILABLE;

begin
  if NOT(SCHEDULE_INPUTS_LIST.NON_EMPTY(T)) then
    SCHEDULE_INPUTS_LIST.ADD(TARGET, MAY_BE_AVAILABLE);
  else
```

```
      if TARGET.THE_UPPER < SCHEDULE_INPUTS_LIST.VALUE(T).THE_UPPER then
        SCHEDULE_INPUTS_LIST.ADD(TARGET, MAY_BE_AVAILABLE);
      elsif NOT(SCHEDULE_INPUTS_LIST.MEMBER(TARGET, MAY_BE_AVAILABLE)) then
        while SCHEDULE_INPUTS_LIST.NON_EMPTY(T) loop
          if TARGET.THE_UPPER > SCHEDULE_INPUTS_LIST.VALUE(T).THE_UPPER then
            PREV := T;
            SCHEDULE_INPUTS_LIST.NEXT(T);
          else
            exit;
          end if;
        end loop;
        SCHEDULE_INPUTS_LIST.ADD(TARGET, T);
        if SCHEDULE_INPUTS_LIST.NON_EMPTY(PREV) then
          PREV.NEXT := T;
        else
          MAY_BE_AVAILABLE := T;
        end if;
      end if;
    end if;
  end EDL_INSERT;
------------------------------------------------
  function OPERATOR_IN_LIST(OPT_ID    : in OPERATOR_ID;
                            IN_LIST   : in SCHEDULE_INPUTS_LIST.LIST)
                                                            return BOOLEAN is
  -- this is used to check if the operators in successors list are already in
  -- the complete MAY_BE_AVAILABLE list for both EST and EDL algorithms.
  TEMP : OPERATOR_ID;
  L    : SCHEDULE_INPUTS_LIST.LIST := IN_LIST;

  begin
    while SCHEDULE_INPUTS_LIST.NON_EMPTY(L) loop
      TEMP := SCHEDULE_INPUTS_LIST.VALUE(L).THE_OPERATOR;
      if  VARSTRING.EQUAL(TEMP, OPT_ID) then
        return TRUE;
      else
        SCHEDULE_INPUTS_LIST.NEXT(L);
      end if;
    end loop;
    return FALSE;
  end OPERATOR_IN_LIST;
------------------------------------------------
  procedure EST_INSERT_SUCCESSORS_OF_OPT
                      (THE_NODE         : in OP_INFO;
                       STOP_TIME        : in VALUE;
                       MAY_BE_AVAILABLE : in out SCHEDULE_INPUTS_LIST.LIST) is
    -- inserts the successors of the selected operator into MAY_BE_AVAILABLE
    -- list in their orders if they do not exist in the list.
    S      : DIGRAPH.V_LISTS.LIST := THE_NODE.SUCCESSORS;
    T      : OPERATOR;
    OPT    : OPERATOR := THE_NODE.NODE;
    TARGET : SCHEDULE_INPUTS;
  begin
    while DIGRAPH.V_LISTS.NON_EMPTY(S) loop
      T := DIGRAPH.V_LISTS.VALUE(S);
      if NOT(OPERATOR_IN_LIST(T.THE_OPERATOR_ID, MAY_BE_AVAILABLE)) then
        TARGET.THE_OPERATOR := DIGRAPH.V_LISTS.VALUE(S).THE_OPERATOR_ID;
        TARGET.THE_LOWER := STOP_TIME;
        EST_INSERT(TARGET, MAY_BE_AVAILABLE);
      end if;
      DIGRAPH.V_LISTS.NEXT(S);
    end loop;
  end EST_INSERT_SUCCESSORS_OF_OPT;
------------------------------------------------
  procedure EDL_INSERT_SUCCESSORS_OF_OPT
                      (THE_NODE         : in OP_INFO;
                       STOP_TIME        : in VALUE;
                       COMPLETE_LIST    : in out SCHEDULE_INPUTS_LIST.LIST;
                       MAY_BE_AVAILABLE : in out SCHEDULE_INPUTS_LIST.LIST) is
    -- inserts the successors of the selected operator into MAY_BE_AVAILABLE
    -- list in their orders if they do not exist in the list.
    S      : DIGRAPH.V_LISTS.LIST := THE_NODE.SUCCESSORS;
    T      : OPERATOR;
    OPT    : OPERATOR := THE_NODE.NODE;
    TARGET : SCHEDULE_INPUTS;
  begin
    while DIGRAPH.V_LISTS.NON_EMPTY(S) loop
      T := DIGRAPH.V_LISTS.VALUE(S);
      if NOT(OPERATOR_IN_LIST(T.THE_OPERATOR_ID, COMPLETE_LIST)) then
        TARGET.THE_OPERATOR := T.THE_OPERATOR_ID;
        TARGET.THE_LOWER := STOP_TIME;
        -- while we are adding the successors, the deadline of these operators
        -- are calculated by adding either their finish_within if exists, or
        -- period to the stop_time of the last operator.
        if T.THE_WITHIN /= 0 then
          TARGET.THE_UPPER := STOP_TIME + T.THE_WITHIN;
        else
          TARGET.THE_UPPER := STOP_TIME + T.THE_PERIOD;
        end if;
        EDL_INSERT(TARGET, MAY_BE_AVAILABLE);
      end if;
      DIGRAPH.V_LISTS.NEXT(S);
    end loop;
  end EDL_INSERT_SUCCESSORS_OF_OPT;
------------------------------------------------
  procedure PROCESS_EST_AGENDA(THE_OP_INFO_LIST: in OP_INFO_LIST.LIST;
                           MAY_BE_AVAILABLE: in out SCHEDULE_INPUTS_LIST.LIST;
                           AGENDA           : in out SCHEDULE_INPUTS_LIST.LIST;
                           HARMONIC_BLOCK_LENGTH : in INTEGER) is

    -- process the MAY_BE_AVILABLE list to produce AGENDA list which is used to
    -- create a schedule for Earliest Start Scheduling Algorithm.
    V         : SCHEDULE_INPUTS_LIST.LIST := MAY_BE_AVAILABLE;
  A          : SCHEDULE_INPUTS_LIST.LIST;
  TEMP       : OP_INFO_LIST.LIST;
  TARGET     : SCHEDULE_INPUTS;
  NEW_INPUT  : SCHEDULE_INPUTS;
  THE_NODE   : OP_INFO;
  CONTINUE   : BOOLEAN;
  STOP_TIME  : VALUE := 0;
  OPT        : SCHEDULE_INPUTS;
  EST        : INTEGER;
  package INTEGERIO is new TEXT_IO.INTEGER_IO(INTEGER);

begin
  while SCHEDULE_INPUTS_LIST.VALUE(V).THE_LOWER < HARMONIC_BLOCK_LENGTH loop
    --no need to check if all the predicessors are in the AGENDA, because this
    -- is the first node and has no predicessors.
    OPT := SCHEDULE_INPUTS_LIST.VALUE(V);
    TEMP := FIND_OPERATOR(THE_OP_INFO_LIST, OPT.THE_OPERATOR);
    THE_NODE := OP_INFO_LIST.VALUE(TEMP);
    if OPT.THE_LOWER > 0 then
      CONTINUE := CHECK_AGENDA(THE_NODE, AGENDA);
    else
      CONTINUE := TRUE;
    end if;

    -- if the opt.is not an end node check if all its successors in AGENDA.
    -- if not, select the other operator and repeat the same procedure.
    while NOT CONTINUE loop
      SCHEDULE_INPUTS_LIST.NEXT(V);
      OPT := SCHEDULE_INPUTS_LIST.VALUE(V);
      TEMP := FIND_OPERATOR(THE_OP_INFO_LIST, OPT.THE_OPERATOR);
      THE_NODE := OP_INFO_LIST.VALUE(TEMP);
      if OPT.THE_LOWER > 0 then
        CONTINUE := CHECK_AGENDA(THE_NODE, AGENDA);
      else
        CONTINUE := TRUE;
      end if;
    end loop;
    TARGET := SCHEDULE_INPUTS_LIST.VALUE(V);
    SCHEDULE_INPUTS_LIST.REMOVE(TARGET, MAY_BE_AVAILABLE);
    Exception_Operator := TARGET.THE_OPERATOR;
    VERIFY_TIME_LEFT(HARMONIC_BLOCK_LENGTH, STOP_TIME);
    if TARGET.THE_LOWER > STOP_TIME then
      TARGET.THE_START := TARGET.THE_LOWER; --zero initially for the first one
    else
      TARGET.THE_START := STOP_TIME;
    end if;
    STOP_TIME := TARGET.THE_START + THE_NODE.NODE.THE_MET;
    TARGET.THE_STOP := STOP_TIME;
    SCHEDULE_INPUTS_LIST.ADD(TARGET, AGENDA);
    EST := TARGET.THE_START + THE_NODE.NODE.THE_PERIOD;
    NEW_INPUT.THE_OPERATOR := TARGET.THE_OPERATOR;
```

```
      NEW_INPUT.THE_LOWER := EST;
      EST_INSERT(NEW_INPUT, MAY_BE_AVAILABLE);
      EST_INSERT_SUCCESSORS_OF_OPT(THE_NODE, STOP_TIME, MAY_BE_AVAILABLE);
      V := MAY_BE_AVAILABLE;
    end loop;
    A := AGENDA;
    SCHEDULE_INPUTS_LIST.LIST_REVERSE(A, AGENDA);
  end PROCESS_EST_AGENDA;
--------------------------------------------------------------------
  procedure PROCESS_EDL_AGENDA(THE_OP_INFO_LIST: in OP_INFO_LIST.LIST;
                           COMPLETE_LIST    : in out SCHEDULE_INPUTS_LIST.LIST;
                           AGENDA           : in out SCHEDULE_INPUTS_LIST.LIST;
                           HARMONIC_BLOCK_LENGTH : in INTEGER) is

    -- process the MAY_BE_AVILABLE list to produce AGENDA list which is used to
    -- create a schedule for Earliest Deadline Scheduling Algorithm.
    V          : SCHEDULE_INPUTS_LIST.LIST := COMPLETE_LIST;
    TEMP       : SCHEDULE_INPUTS_LIST.LIST := COMPLETE_LIST;
    A          : SCHEDULE_INPUTS_LIST.LIST;
    T          : OP_INFO_LIST.LIST;
    PREV       : SCHEDULE_INPUTS_LIST.LIST := null;
    TARGET     : SCHEDULE_INPUTS;
    NEW_INPUT  : SCHEDULE_INPUTS;
    THE_NODE   : OP_INFO;
    CONTINUE   : BOOLEAN;
    STOP_TIME  : VALUE := 0;
    OPT        : SCHEDULE_INPUTS;
    EST        : INTEGER;
    package INTEGERIO is new TEXT_IO.INTEGER_IO(INTEGER);

  begin
    while SCHEDULE_INPUTS_LIST.NON_EMPTY(TEMP) loop
      if SCHEDULE_INPUTS_LIST.VALUE(TEMP).THE_LOWER < HARMONIC_BLOCK_LENGTH then
        -- no need to check if all the predicessors are in the AGENDA, because
        -- for the first node there is no predicessors.
        OPT := SCHEDULE_INPUTS_LIST.VALUE(V);
        T   := FIND_OPERATOR(THE_OP_INFO_LIST, OPT.THE_OPERATOR);
        THE_NODE := OP_INFO_LIST.VALUE(T);
        if OPT.THE_LOWER > 0 then

          -- when the earliest start time of the operator is not zero, we need
          -- to check if all the predicessors of the operator are in AGENDA. No
          -- check otherwise.
          CONTINUE := CHECK_AGENDA(THE_NODE, AGENDA);
        else
          CONTINUE := TRUE;
        end if;

        -- if the opt.is not an end node check if all its successors in AGENDA.
        -- if not, select the other operator and repeat the same procedure.
        while NOT CONTINUE loop
      SCHEDULE_INPUTS_LIST.NEXT(V);
      OPT := SCHEDULE_INPUTS_LIST.VALUE(V);
      T   := FIND_OPERATOR(THE_OP_INFO_LIST, OPT.THE_OPERATOR);
      THE_NODE := OP_INFO_LIST.VALUE(T);
      if OPT.THE_LOWER > 0 then
        CONTINUE := CHECK_AGENDA(THE_NODE, AGENDA);
      else
        CONTINUE := TRUE;
      end if;
    end loop;
    TARGET := SCHEDULE_INPUTS_LIST.VALUE(V);
    SCHEDULE_INPUTS_LIST.REMOVE(TARGET, TEMP);
    if SCHEDULE_INPUTS_LIST.NON_EMPTY(PREV) then
      PREV.NEXT := TEMP;
    else
      COMPLETE_LIST := TEMP;
    end if;
  Exception_Operator := TARGET.THE_OPERATOR;
  VERIFY_TIME_LEFT(HARMONIC_BLOCK_LENGTH, STOP_TIME);
  if TARGET.THE_LOWER > STOP_TIME then
    TARGET.THE_START := TARGET.THE_LOWER; --zero initially for the first one
  else
    TARGET.THE_START := STOP_TIME;
  end if;
  STOP_TIME := TARGET.THE_START + THE_NODE.NODE.THE_MET;
  TARGET.THE_STOP := STOP_TIME;
  SCHEDULE_INPUTS_LIST.ADD(TARGET, AGENDA);
  EST := TARGET.THE_START + THE_NODE.NODE.THE_PERIOD;
  NEW_INPUT.THE_OPERATOR := TARGET.THE_OPERATOR;
  NEW_INPUT.THE_LOWER := EST;

  if THE_NODE.NODE.THE_WITHIN /= 0 then
    NEW_INPUT.THE_UPPER := EST + THE_NODE.NODE.THE_WITHIN;
  else
    NEW_INPUT.THE_UPPER := EST + THE_NODE.NODE.THE_PERIOD;
  end if;
  EDL_INSERT(NEW_INPUT, TEMP);

  -- this is to keep track of the COMPLETE_LIST pointer
  if SCHEDULE_INPUTS_LIST.NON_EMPTY(PREV) then
    -- the pointer is pointing a record other than first one.
    PREV.NEXT := TEMP;
  else
    -- the pointer is pointing the first record in the list.
    COMPLETE_LIST := TEMP;
  end if;

  EDL_INSERT_SUCCESSORS_OF_OPT(THE_NODE, STOP_TIME, COMPLETE_LIST, TEMP);
  V := TEMP;

  -- this is to keep track of the COMPLETE_LIST pointer
```

```
      if SCHEDULE_INPUTS_LIST.NON_EMPTY(PREV) then
        -- the pointer is pointing a record other than first one.
        PREV.NEXT := TEMP;
      else
        -- the pointer is pointing the first record in the list.
        COMPLETE_LIST := TEMP;
      end if;

    else
      PREV := TEMP;
      SCHEDULE_INPUTS_LIST.NEXT(TEMP);
      V := TEMP;
    end if;
    end loop;
    -- If any operator is missed to be scheduled then exception MISSED_OPERATOR
    -- is raised.
    while SCHEDULE_INPUTS_LIST.NON_EMPTY(TEMP) loop
      if not(OPERATOR_IN_LIST(SCHEDULE_INPUTS_LIST.VALUE(TEMP).THE_OPERATOR,
                                                               AGENDA)) then
        Exception_Operator := SCHEDULE_INPUTS_LIST.VALUE(TEMP).THE_OPERATOR;
        raise MISSED_OPERATOR;
      endif;
      SCHEDULE_INPUTS_LIST.NEXT(TEMP);
    end loop;
    A := AGENDA;
    SCHEDULE_INPUTS_LIST.LIST_REVERSE(A, AGENDA);
  end PROCESS_EDL_AGENDA;
-----------------------------------------------------------------------------
  procedure SCHEDULE_WITH_EARLIEST_START(THE_GRAPH : in DIGRAPH.GRAPH;
                                        AGENDA : in out SCHEDULE_INPUTS_LIST.LIST;
                                        HARMONIC_BLOCK_LENGTH : in INTEGER) is
    -- used to find a feasible schedule for Earliest Start Scheduling Algorithm.
    THE_OP_INFO_LIST : OP_INFO_LIST.LIST;
    MAY_BE_AVAILABLE : SCHEDULE_INPUTS_LIST.LIST;
    H_B_L : INTEGER := HARMONIC_BLOCK_LENGTH;
    L : OP_INFO_LIST.LIST;
    P : OP_INFO;

  begin
    BUILD_OP_INFO_LIST(THE_GRAPH, THE_OP_INFO_LIST);
    L := THE_OP_INFO_LIST;
    -- put all the end nodes, which has no predicessors, into MAY_BE_AVAILABLE
    -- list
    while OP_INFO_LIST.NON_EMPTY(L) loop
      P := OP_INFO_LIST.VALUE(L);
      if NOT(DIGRAPH.V_LISTS.NON_EMPTY(P.PREDICESSORS)) then
        PROCESS_EST_END_NODE(MAY_BE_AVAILABLE, P.NODE);
      end if;
      OP_INFO_LIST.NEXT(L);
    end loop;
    PROCESS_EST_AGENDA(THE_OP_INFO_LIST, MAY_BE_AVAILABLE, AGENDA, H_B_L);
  end SCHEDULE_WITH_EARLIEST_START;
-----------------------------------------------------------------------------
  procedure SCHEDULE_WITH_EARLIEST_DEADLINE(THE_GRAPH : in DIGRAPH.GRAPH;
                           AGENDA                  : in out SCHEDULE_INPUTS_LIST.LIST;
                           HARMONIC_BLOCK_LENGTH : in INTEGER) is

    -- used to find a feasible schedule for Earliest Deadline Scheduling
    -- Algorithm
    THE_OP_INFO_LIST : OP_INFO_LIST.LIST;
    MAY_BE_AVAILABLE : SCHEDULE_INPUTS_LIST.LIST;
    H_B_L : INTEGER := HARMONIC_BLOCK_LENGTH;
    L : OP_INFO_LIST.LIST;
    P : OP_INFO;

  begin
    BUILD_OP_INFO_LIST(THE_GRAPH, THE_OP_INFO_LIST);
    L := THE_OP_INFO_LIST;
    -- put all the end nodes, which has no predicessors, into MAY_BE_AVAILABLE
    -- list
    while OP_INFO_LIST.NON_EMPTY(L) loop
      P := OP_INFO_LIST.VALUE(L);
      if NOT(DIGRAPH.V_LISTS.NON_EMPTY(P.PREDICESSORS)) then
        PROCESS_EDL_END_NODE(MAY_BE_AVAILABLE, P.NODE);
      end if;
      OP_INFO_LIST.NEXT(L);
    end loop;
    PROCESS_EDL_AGENDA(THE_OP_INFO_LIST, MAY_BE_AVAILABLE, AGENDA, H_B_L);
  end SCHEDULE_WITH_EARLIEST_DEADLINE;
--------------------------------------------------------------
  procedure CREATE_STATIC_SCHEDULE (THE_GRAPH : in DIGRAPH.GRAPH;
                           THE_SCHEDULE_INPUTS    : in SCHEDULE_INPUTS_LIST.LIST;
                           HARMONIC_BLOCK_LENGTH : in INTEGER) is
    -- creates the static schedule output and prints to "ss.a" file.
    V_LIST : DIGRAPH.V_LISTS.LIST := THE_GRAPH.VERTICES;
    S : SCHEDULE_INPUTS_LIST.LIST := THE_SCHEDULE_INPUTS;
    SCHEDULE   : TEXT_IO.FILE_TYPE;
    OUTPUT : TEXT_IO.FILE_MODE := TEXT_IO.OUT_FILE;
    COUNTER : INTEGER := 1;

    package VALUE_IO is new TEXT_IO.INTEGER_IO(VALUE);
    use VALUE_IO;
    package INTEGERIO is new TEXT_IO.INTEGER_IO(INTEGER);
    use INTEGERIO;
    package FLOATIO is new TEXT_IO.FLOAT_IO(FLOAT);
    use FLOATIO;

  begin
    TEXT_IO.CREATE(SCHEDULE, OUTPUT, "ss.a");
    TEXT_IO.PUT_LINE(SCHEDULE, "with TL; use TL;");
    TEXT_IO.PUT_LINE(SCHEDULE, "with DS_PACKAGE; use DS_PACKAGE;");
    TEXT_IO.PUT(SCHEDULE, "with PRIORITY_DEFINITIONS; ");
TEXT_IO.PUT_LINE (SCHEDULE, "use PRIORITY_DEFINITIONS;");
TEXT_IO.PUT_LINE(SCHEDULE, "with CALENDAR; use CALENDAR;");
TEXT_IO.PUT_LINE(SCHEDULE, "with TEXT_IO; use TEXT_IO;");
TEXT_IO.PUT_LINE(SCHEDULE, "procedure STATIC_SCHEDULE is");

while DIGRAPH.V_LISTS.NON_EMPTY(V_LIST) loop
  TEXT_IO.SET_COL(SCHEDULE, 3);
  VARSTRING.PUT(SCHEDULE, DIGRAPH.V_LISTS.VALUE(V_LIST).THE_OPERATOR_ID);
  TEXT_IO.PUT_LINE(SCHEDULE, "_TIMING_ERROR : exception;");
  DIGRAPH.V_LISTS.NEXT(V_LIST);
end loop;

TEXT_IO.SET_COL(SCHEDULE, 3);
TEXT_IO.PUT_LINE(SCHEDULE, "task SCHEDULE is");
TEXT_IO.SET_COL(SCHEDULE, 5);
TEXT_IO.PUT_LINE(SCHEDULE, "pragma priority (STATIC_SCHEDULE_PRIORITY);");
TEXT_IO.SET_COL(SCHEDULE, 3);
TEXT_IO.PUT_LINE(SCHEDULE, "end SCHEDULE;");
TEXT_IO.NEW_LINE(SCHEDULE);
TEXT_IO.SET_COL(SCHEDULE, 3);
TEXT_IO.PUT_LINE(SCHEDULE, "task body SCHEDULE is");
TEXT_IO.PUT(SCHEDULE, "    PERIOD : constant := ");
INTEGERIO.PUT(SCHEDULE, HARMONIC_BLOCK_LENGTH, 1);
TEXT_IO.PUT_LINE(SCHEDULE, ";");
S := THE_SCHEDULE_INPUTS;
while SCHEDULE_INPUTS_LIST.NON_EMPTY(S) loop
  TEXT_IO.SET_COL(SCHEDULE, 5);
  VARSTRING.PUT(SCHEDULE, SCHEDULE_INPUTS_LIST.VALUE(S).THE_OPERATOR);
  TEXT_IO.PUT(SCHEDULE, "_STOP_TIME");
  INTEGERIO.PUT(SCHEDULE, COUNTER,1);
  TEXT_IO.PUT(SCHEDULE, " : constant := ");
  FLOATIO.PUT(SCHEDULE, FLOAT(SCHEDULE_INPUTS_LIST.VALUE(S).THE_STOP),3,1,0);
  TEXT_IO.PUT_LINE(SCHEDULE, ";");
  SCHEDULE_INPUTS_LIST.NEXT(S);
  COUNTER := COUNTER + 1;
end loop;
TEXT_IO.SET_COL(SCHEDULE, 5);
TEXT_IO.PUT_LINE(SCHEDULE, "SLACK_TIME : duration;");
TEXT_IO.SET_COL(SCHEDULE, 5);
TEXT_IO.PUT_LINE(SCHEDULE, "START_OF_PERIOD : time := clock;");
TEXT_IO.PUT_LINE(SCHEDULE, "begin");
TEXT_IO.PUT_LINE(SCHEDULE, "  loop");
TEXT_IO.SET_COL(SCHEDULE, 5);
TEXT_IO.PUT(SCHEDULE, "begin");

S := THE_SCHEDULE_INPUTS;
COUNTER := 1;
while SCHEDULE_INPUTS_LIST.NON_EMPTY(S) loop
  TEXT_IO.SET_COL(SCHEDULE, 7);
  VARSTRING.PUT(SCHEDULE, SCHEDULE_INPUTS_LIST.VALUE(S).THE_OPERATOR);
  TEXT_IO.PUT_LINE(SCHEDULE, ";");
```

```
      TEXT_IO.SET_COL(SCHEDULE, 7);
      TEXT_IO.PUT(SCHEDULE, "SLACK_TIME := START_OF_PERIOD + ");
      VARSTRING.PUT(SCHEDULE, SCHEDULE_INPUTS_LIST.VALUE(S).THE_OPERATOR);
      TEXT_IO.PUT(SCHEDULE, "_STOP_TIME");
      INTEGERIO.PUT(SCHEDULE, COUNTER,1);
      TEXT_IO.PUT_LINE(SCHEDULE, " - CLOCK;");
      TEXT_IO.SET_COL(SCHEDULE, 7);
      TEXT_IO.PUT_LINE(SCHEDULE, "if SLACK_TIME >= 0.0 then");
      TEXT_IO.SET_COL(SCHEDULE, 9);
      TEXT_IO.PUT_LINE(SCHEDULE, "delay (SLACK_TIME);");
      TEXT_IO.SET_COL(SCHEDULE, 7);
      TEXT_IO.PUT_LINE(SCHEDULE, "else");
      TEXT_IO.SET_COL(SCHEDULE, 9);
      TEXT_IO.PUT(SCHEDULE, "raise ");
      VARSTRING.PUT(SCHEDULE, SCHEDULE_INPUTS_LIST.VALUE(S).THE_OPERATOR);
      TEXT_IO.PUT_LINE(SCHEDULE, "_TIMING_ERROR;");
      TEXT_IO.SET_COL(SCHEDULE, 7);
      TEXT_IO.PUT_LINE(SCHEDULE, "end if;");
      SCHEDULE_INPUTS_LIST.NEXT(S);
      if SCHEDULE_INPUTS_LIST.NON_EMPTY(S) then
        -- pointer is pointing to the next record after this.
        TEXT_IO.SET_COL(SCHEDULE, 7);
        TEXT_IO.PUT(SCHEDULE, "delay (START_OF_PERIOD + ");
        FLOATIO.PUT(SCHEDULE,FLOAT(SCHEDULE_INPUTS_LIST.VALUE(S).THE_START),3,1,0
        TEXT_IO.PUT_LINE(SCHEDULE, " - CLOCK);");
        TEXT_IO.NEW_LINE(SCHEDULE);
      end if;
      COUNTER := COUNTER + 1;
    end loop;

    TEXT_IO.SET_COL(SCHEDULE, 7);
    TEXT_IO.PUT_LINE(SCHEDULE, "START_OF_PERIOD := START_OF_PERIOD + PERIOD;");
    TEXT_IO.SET_COL(SCHEDULE, 7);
    TEXT_IO.PUT_LINE(SCHEDULE, "delay (START_OF_PERIOD - clock);");

    TEXT_IO.SET_COL(SCHEDULE, 7);
    TEXT_IO.PUT_LINE(SCHEDULE, "exception");
    V_LIST := THE_GRAPH.VERTICES;
    while DIGRAPH.V_LISTS.NON_EMPTY(V_LIST) loop
      TEXT_IO.SET_COL(SCHEDULE, 9);
      TEXT_IO.PUT(SCHEDULE, "when ");
      VARSTRING.PUT(SCHEDULE, DIGRAPH.V_LISTS.VALUE(V_LIST).THE_OPERATOR_ID);
      TEXT_IO.PUT_LINE(SCHEDULE, "_TIMING_ERROR =>");
      TEXT_IO.SET_COL(SCHEDULE, 11);
      TEXT_IO.PUT(SCHEDULE, "PUT_LINE(""timing error from operator ");
      VARSTRING.PUT(SCHEDULE, DIGRAPH.V_LISTS.VALUE(V_LIST).THE_OPERATOR_ID);
      TEXT_IO.PUT_LINE(SCHEDULE, """);");
      TEXT_IO.SET_COL(SCHEDULE, 11);
      TEXT_IO.PUT_LINE(SCHEDULE, "START_OF_PERIOD := clock;");
      DIGRAPH.V_LISTS.NEXT(V_LIST);
    end loop;
    TEXT_IO.SET_COL(SCHEDULE, 7);
    TEXT_IO.PUT_LINE(SCHEDULE, "end;");
    TEXT_IO.SET_COL(SCHEDULE, 5);
    TEXT_IO.PUT_LINE(SCHEDULE, "end loop;");
    TEXT_IO.SET_COL(SCHEDULE, 3);
    TEXT_IO.PUT_LINE(SCHEDULE, "end SCHEDULE;");
    TEXT_IO.NEW_LINE(SCHEDULE);
    TEXT_IO.PUT_LINE(SCHEDULE, "begin");
    TEXT_IO.SET_COL(SCHEDULE, 3);
    TEXT_IO.PUT_LINE(SCHEDULE, "null;");
    TEXT_IO.PUT_LINE(SCHEDULE, "end STATIC_SCHEDULE;");

  end CREATE_STATIC_SCHEDULE;

end OPERATOR_SCHEDULER;
```

```
--=============================================================================
-- EXCEPTION_HANDLER - "e_handler_s.a, e_handler_b"; handles most of the
--                     exceptions,and inform the user about the situation.
--=============================================================================
with FILES; use FILES;

package EXCEPTION_HANDLER is

  procedure CRIT_O_L_MET(Exception_Operator : in OPERATOR_ID);

  procedure MET_N_L_T_PERIOD(Exception_Operator : in OPERATOR_ID);

  procedure MET_N_L_T_MRT(Exception_Operator : in OPERATOR_ID);

  procedure MCP_N_L_T_MRT(Exception_Operator : in OPERATOR_ID);

  procedure MCP_L_T_MET(Exception_Operator : in OPERATOR_ID);

  procedure MET_I_G_T_FINISH_WITHIN(Exception_Operator : in OPERATOR_ID);

  procedure PERIOD_L_T_FINISH_WITHIN(Exception_Operator : in OPERATOR_ID);

  procedure SPORADIC_O_L_MCP(Exception_Operator : in OPERATOR_ID);

  procedure SPORADIC_O_L_MRT(Exception_Operator : in OPERATOR_ID);

  procedure S_I_L_BAD_VALUE;

  procedure V_L_BAD_VALUE;

  procedure E_L_BAD_VALUE;

  procedure NO_B_BLOCK;

  procedure NO_OP_IN_LIST;

end EXCEPTION_HANDLER;
--------------------------
with TEXT_IO;
with FILES; use FILES;

package body EXCEPTION_HANDLER is

  procedure CRIT_O_L_MET(Exception_Operator : in OPERATOR_ID) is
  begin
    TEXT_IO.PUT ("Critical Operator ");
    VARSTRING.PUT (Exception_Operator);
    TEXT_IO.PUT_LINE (" must have an MET");
  end CRIT_O_L_MET;

  procedure MET_N_L_T_PERIOD(Exception_Operator : in OPERATOR_ID) is
```

```
begin
  TEXT_IO.PUT ("MET is greater than PERIOD in operator ");
  VARSTRING.PUT_LINE (Exception_Operator);
end MET_N_L_T_PERIOD;

procedure MET_N_L_T_MRT(Exception_Operator : in OPERATOR_ID) is
begin
  TEXT_IO.PUT ("MET is greater than MRT in operator ");
  VARSTRING.PUT_LINE (Exception_Operator);
end MET_N_L_T_MRT;

procedure MCP_N_L_T_MRT(Exception_Operator : in OPERATOR_ID) is
begin
  TEXT_IO.PUT ("MCP is greater than MRT in operator ");
  VARSTRING.PUT_LINE (Exception_Operator);
end MCP_N_L_T_MRT;

procedure MCP_L_T_MET(Exception_Operator : in OPERATOR_ID) is
begin
  TEXT_IO.PUT ("MCP is less than MET in operator ");
  VARSTRING.PUT_LINE (Exception_Operator);
end MCP_L_T_MET;

procedure MET_I_G_T_FINISH_WITHIN(Exception_Operator : in OPERATOR_ID) is
begin
  TEXT_IO.PUT ("MET is greater than FINISH_WITHIN in operator ");
  VARSTRING.PUT_LINE (Exception_Operator);
end MET_I_G_T_FINISH_WITHIN;

procedure PERIOD_L_T_FINISH_WITHIN(Exception_Operator : in OPERATOR_ID) is
begin
  TEXT_IO.PUT ("PERIOD is less than FINISH_WITHIN in operator ");
  VARSTRING.PUT_LINE (Exception_Operator);
end PERIOD_L_T_FINISH_WITHIN;

procedure SPORADIC_O_L_MCP(Exception_Operator : in OPERATOR_ID) is
begin
  TEXT_IO.PUT ("Sporadic Operator ");
  VARSTRING.PUT (Exception_Operator);
  TEXT_IO.PUT_LINE (" must have an MCP");
end SPORADIC_O_L_MCP;

procedure SPORADIC_O_L_MRT(Exception_Operator : in OPERATOR_ID) is
begin
  TEXT_IO.PUT ("Sporadic Operator ");
  VARSTRING.PUT (Exception_Operator);
  TEXT_IO.PUT_LINE (" must have an MRT");
end SPORADIC_O_L_MRT;

procedure S_I_L_BAD_VALUE is
begin
```

```
    TEXT_IO.PUT ("You try to get a schedule input where your pointer ");
    TEXT_IO.PUT_LINE ("is pointing a null record.");
  end S_I_L_BAD_VALUE;

  procedure V_L_BAD_VALUE is
  begin
    TEXT_IO.PUT ("You try to get an operator where your pointer ");
    TEXT_IO.PUT_LINE ("is pointing a null record.");
  end V_L_BAD_VALUE;

  procedure E_L_BAD_VALUE is
  begin
    TEXT_IO.PUT ("You try to get a link data where your pointer ");
    TEXT_IO.PUT_LINE ("is pointing a null record.");
  end E_L_BAD_VALUE;

  procedure NO_B_BLOCK is
  begin
    TEXT_IO.PUT_LINE ("There is no BASE BLOCK in this system.");
  end NO_B_BLOCK;

 procedure NO_OP_IN_LIST is
  begin
    TEXT_IO.PUT_LINE ("There is no CRITICAL OPERATOR in this system.");
  end NO_OP_IN_LIST;

end EXCEPTION_HANDLER;
```

```
--===========================================================================
-- STATIC_SCHEDULER - "driver.a"; this is the driver program of the Menu driven
                      standalone scheduler.
--===========================================================================
with TEXT_IO;
with FILES; use FILES;
with FILE_PROCESSOR;
with EXCEPTION_HANDLER;
with TOPOLOGICAL_SORTER;
with HARMONIC_BLOCK_BUILDER;
with OPERATOR_SCHEDULER;

procedure STATIC_SCHEDULER is

  THE_GRAPH       : DIGRAPH.GRAPH;
  PRECEDENCE_LIST : DIGRAPH.V_LISTS.LIST;
  SCH_INPUTS      : SCHEDULE_INPUTS_LIST.LIST;
  AGENDA          : SCHEDULE_INPUTS_LIST.LIST;
  BASE_BLOCK      : INTEGER;
  H_B_LENGTH      : INTEGER;
  STOP_TIME       : INTEGER := 0;
  CHOICE          : INTEGER;

  procedure MAIN_MENU is
  begin
   TEXT_IO.NEW_PAGE;
   TEXT_IO.NEW_LINE(4);
   TEXT_IO.PUT_LINE("                              MAIN MENU");
   TEXT_IO.PUT_LINE("                              ---------");
   TEXT_IO.NEW_LINE(2);
   TEXT_IO.PUT_LINE("              1) THE HARMONIC BLOCK WITH PRECEDENCE");
   TEXT_IO.PUT_LINE("                 CONSTRAINTS SCHEDULING ALGORITHM");
   TEXT_IO.NEW_LINE;
   TEXT_IO.PUT_LINE("              2) THE EARLIEST START SCHEDULING ALGORITHM");
   TEXT_IO.NEW_LINE;
   TEXT_IO.PUT_LINE("              3) THE EARLIEST DEADLINE SCHEDULING ALGORITHM");
   TEXT_IO.NEW_LINE;
   TEXT_IO.PUT_LINE("              4) EXIT");
   TEXT_IO.NEW_LINE(2);
  end MAIN_MENU;

begin
  FILE_PROCESSOR.SEPARATE_DATA (THE_GRAPH);
  FILE_PROCESSOR.VALIDATE_DATA(THE_GRAPH);
  HARMONIC_BLOCK_BUILDER.CALC_PERIODIC_EQUIVALENTS (THE_GRAPH);
  TOPOLOGICAL_SORTER.TOPOLOGICAL_SORT(THE_GRAPH, PRECEDENCE_LIST);
  HARMONIC_BLOCK_BUILDER.FIND_BASE_BLOCK (PRECEDENCE_LIST, BASE_BLOCK);
  HARMONIC_BLOCK_BUILDER.FIND_BLOCK_LENGTH (PRECEDENCE_LIST,H_B_LENGTH);
  OPERATOR_SCHEDULER.TEST_DATA(PRECEDENCE_LIST, H_B_LENGTH);
  TEXT_IO.PUT_LINE("passed TEST_DATA");
  loop
```

```
declare
  WRONG_ENTRY : exception;
begin
MAIN_MENU;
if NOT(TEST_VERIFIED) then
  TEXT_IO.PUT("Although a schedule may be possible, there is no ");
  TEXT_IO.PUT_LINE("guarantee that it will execute");
  TEXT_IO.PUT_LINE("within the required timing constraints.");
  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT_LINE("IF YOU STILL WANT TO RUN THE ALGORITHMS !");
end if;
TEXT_IO.PUT("      Enter your choice (1/2/3) :");
INTEGERIO.GET(CHOICE);

case CHOICE is
  when 1 =>
    begin
     TEXT_IO.NEW_PAGE;
     OPERATOR_SCHEDULER.SCHEDULE_INITIAL_SET
                               (PRECEDENCE_LIST,SCH_INPUTS,H_B_LENGTH,STOP_TIME);
     TEXT_IO.PUT_LINE("passed SCHEDULE_INITIAL_SET");
     OPERATOR_SCHEDULER.SCHEDULE_REST_OF_BLOCK
                               (PRECEDENCE_LIST,SCH_INPUTS,H_B_LENGTH,STOP_TIME);
     TEXT_IO.PUT_LINE("passed SCHEDULE_REST_OF_BLOCK");
     OPERATOR_SCHEDULER.CREATE_STATIC_SCHEDULE(THE_GRAPH,SCH_INPUTS,H_B_LENGTH)
     TEXT_IO.PUT_LINE("passed CREATE_STATIC_SCHEDULE");
     TEXT_IO.PUT_LINE("A feasible schedule found, READ schedule.out FILE...");
     SCH_INPUTS := null;
     delay 3.0;
    exception
      when OPERATOR_SCHEDULER.MISSED_DEADLINE =>
        TEXT_IO.PUT ("The Operator ");
        VARSTRING.PUT (Exception_Operator);
        TEXT_IO.PUT (" MISSED ITS DEADLINE.");
        delay 5.0;
      when OPERATOR_SCHEDULER.OVER_TIME =>
        TEXT_IO.PUT ("The Operator ");
        VARSTRING.PUT (Exception_Operator);
        TEXT_IO.PUT (" is OVER_TIME.");
        delay 5.0;
    end;

   when 2 =>
     begin
       TEXT_IO.NEW_PAGE;
text_io.put_line("OK I AM IN EARLIEST_START!!!");
       OPERATOR_SCHEDULER.SCHEDULE_WITH_EARLIEST_START
                                                    (THE_GRAPH,AGENDA,H_B_LENGTH);
       OPERATOR_SCHEDULER.CREATE_STATIC_SCHEDULE(THE_GRAPH,AGENDA,H_B_LENGTH);
       TEXT_IO.PUT_LINE("A feasible schedule found, READ schedule.out FILE..."
       AGENDA := null;
      delay 5.0;
    exception
      when OPERATOR_SCHEDULER.MISSED_DEADLINE =>
        TEXT_IO.PUT ("The Operator ");
        VARSTRING.PUT (Exception_Operator);
        TEXT_IO.PUT_LINE (" MISSED ITS DEADLINE.");
        delay 5.0;
      when OPERATOR_SCHEDULER.OVER_TIME =>
        TEXT_IO.PUT ("The Operator ");
        VARSTRING.PUT (Exception_Operator);
        TEXT_IO.PUT_LINE (" is OVER_TIME.");
        delay 5.0;
    end;

  when 3 =>
    begin
     TEXT_IO.NEW_PAGE;
     OPERATOR_SCHEDULER.SCHEDULE_WITH_EARLIEST_DEADLINE
                                         (THE_GRAPH,AGENDA,H_B_LENGTH);
     OPERATOR_SCHEDULER.CREATE_STATIC_SCHEDULE(THE_GRAPH,AGENDA,H_B_LENGTH);
     TEXT_IO.PUT_LINE("A feasible schedule found, READ schedule.out FILE...");
     AGENDA := null;
     delay 3.0;
    exception
      when OPERATOR_SCHEDULER.MISSED_DEADLINE =>
        TEXT_IO.PUT ("The Operator ");
        VARSTRING.PUT (Exception_Operator);
        TEXT_IO.PUT (" MISSED ITS DEADLINE.");
        delay 5.0;
      when OPERATOR_SCHEDULER.OVER_TIME =>
        TEXT_IO.PUT ("The Operator ");
        VARSTRING.PUT (Exception_Operator);
        TEXT_IO.PUT_LINE (" is OVER_TIME.");
        delay 5.0;
      when STATIC_SCHEDULER.MISSED_OPERATOR =>
        TEXT_IO.PUT ("The Operator ");
        VARSTRING.PUT (Exception_Operator);
        TEXT_IO.PUT_LINE (" can not be scheduled in this algorithm.");
        TEXT_IO.PUT_LINE (" There is no feasible solution.");
        delay 5.0;
     end;
   when 4 => exit;

   when others => raise WRONG_ENTRY;
 end case;

exception
 when WRONG_ENTRY =>
   TEXT_IO.PUT_LINE("THE NUMBER ENTERED IS NOT IN MENU !");
   TEXT_IO.PUT_LINE("Please try again...");
   delay 3.0;
```

```
    end;
    end loop;

  exception
    when FILE_PROCESSOR.CRIT_OP_LACKS_MET =>
      EXCEPTION_HANDLER.CRIT_O_L_MET(Exception_Operator);

    when FILE_PROCESSOR.MET_NOT_LESS_THAN_PERIOD =>
      EXCEPTION_HANDLER.MET_N_L_T_PERIOD(Exception_Operator);

    when FILE_PROCESSOR.MET_NOT_LESS_THAN_MRT =>
      EXCEPTION_HANDLER.MET_N_L_T_MRT(Exception_Operator);

    when FILE_PROCESSOR.MCP_NOT_LESS_THAN_MRT =>
      EXCEPTION_HANDLER.MCP_N_L_T_MRT(Exception_Operator);

    when FILE_PROCESSOR.MCP_LESS_THAN_MET =>
      EXCEPTION_HANDLER.MCP_L_T_MET(Exception_Operator);

    when FILE_PROCESSOR.MET_IS_GREATER_THAN_FINISH_WITHIN =>
      EXCEPTION_HANDLER.MET_I_G_T_FINISH_WITHIN(Exception_Operator);

    when FILE_PROCESSOR.PERIOD_LESS_THAN_FINISH_WITHIN =>
      EXCEPTION_HANDLER.PERIOD_L_T_FINISH_WITHIN(Exception_Operator);

    when FILE_PROCESSOR.SPORADIC_OP_LACKS_MCP =>
      EXCEPTION_HANDLER.SPORADIC_O_L_MCP(Exception_Operator);

    when FILE_PROCESSOR.SPORADIC_OP_LACKS_MRT =>
      EXCEPTION_HANDLER.SPORADIC_O_L_MRT(Exception_Operator);

    when SCHEDULE_INPUTS_LIST.BAD_VALUE =>
      EXCEPTION_HANDLER.S_I_L_BAD_VALUE;

    when DIGRAPH.V_LISTS.BAD_VALUE =>
      EXCEPTION_HANDLER.V_L_BAD_VALUE;

    when DIGRAPH.E_LISTS.BAD_VALUE =>
      EXCEPTION_HANDLER.E_L_BAD_VALUE;

    when HARMONIC_BLOCK_BUILDER.NO_BASE_BLOCK =>
      EXCEPTION_HANDLER.NO_B_BLOCK;

    when HARMONIC_BLOCK_BUILDER.NO_OPERATOR_IN_LIST =>
      EXCEPTION_HANDLER.NO_OP_IN_LIST;

    when HARMONIC_BLOCK_BUILDER.MET_NOT_LESS_THAN_PERIOD =>
      EXCEPTION_HANDLER.MET_N_L_T_PERIOD(Exception_Operator);

  end STATIC_SCHEDULER;
```

APPENDIX F. DRIVER PROGRAM USED IN CAPS

```
with TEXT_IO;
with FILES; use FILES;
with FILE_PROCESSOR;
with EXCEPTION_HANDLER;
with TOPOLOGICAL_SORTER;
with HARMONIC_BLOCK_BUILDER;
with OPERATOR_SCHEDULER;

procedure STATIC_SCHEDULER is

  THE_GRAPH         : DIGRAPH.GRAPH;
  PRECEDENCE_LIST : DIGRAPH.V_LISTS.LIST;
  SCH_INPUTS      : SCHEDULE_INPUTS_LIST.LIST;
  AGENDA          : SCHEDULE_INPUTS_LIST.LIST;
  BASE_BLOCK      : INTEGER;
  H_B_LENGTH      : INTEGER;
  STOP_TIME       : INTEGER := 0;

begin
  FILE_PROCESSOR.SEPARATE_DATA (THE_GRAPH);
  FILE_PROCESSOR.VALIDATE_DATA(THE_GRAPH);
  TOPOLOGICAL_SORTER.TOPOLOGICAL_SORT(THE_GRAPH, PRECEDENCE_LIST);
  HARMONIC_BLOCK_BUILDER.CALC_PERIODIC_EQUIVALENTS (PRECEDENCE_LIST);
  HARMONIC_BLOCK_BUILDER.FIND_BASE_BLOCK (PRECEDENCE_LIST, BASE_BLOCK);
  HARMONIC_BLOCK_BUILDER.FIND_BLOCK_LENGTH (PRECEDENCE_LIST,H_B_LENGTH);
  OPERATOR_SCHEDULER.TEST_DATA(PRECEDENCE_LIST, H_B_LENGTH);
  loop
    if NOT(TEST_VERIFIED) then
      TEXT_IO.PUT("Although a schedule may be possible, there is no ");
      TEXT_IO.PUT_LINE("guarantee that it will execute");
      TEXT_IO.PUT_LINE("within the required timing constraints.");
      TEXT_IO.NEW_LINE;
    end if;
    begin
      OPERATOR_SCHEDULER.SCHEDULE_INITIAL_SET
                          (PRECEDENCE_LIST,SCH_INPUTS,H_B_LENGTH,STOP_TIME);
      OPERATOR_SCHEDULER.SCHEDULE_REST_OF_BLOCK
                          (PRECEDENCE_LIST,SCH_INPUTS,H_B_LENGTH,STOP_TIME);
      OPERATOR_SCHEDULER.CREATE_STATIC_SCHEDULE
                                  (THE_GRAPH,SCH_INPUTS,H_B_LENGTH);
      TEXT_IO.PUT("A feasible schedule found, ");
      TEXT_IO.PUT_LINE("the Harmonic Block with Precedence Constraints ");
      TEXT_IO.PUT_LINE("Scheduling Algorithm Used. ");
      SCH_INPUTS := null;
      exit;
    exception
      when OPERATOR_SCHEDULER.MISSED_DEADLINE =>
      null;
      when OPERATOR_SCHEDULER.OVER_TIME =>
      null;
    end;

    begin
      HARMONIC_BLOCK_BUILDER.CALC_PERIODIC_EQUIVALENTS (THE_GRAPH.VERTICES);
      OPERATOR_SCHEDULER.SCHEDULE_WITH_EARLIEST_START
                                        (THE_GRAPH,AGENDA,H_B_LENGTH);
      OPERATOR_SCHEDULER.CREATE_STATIC_SCHEDULE(THE_GRAPH,AGENDA,H_B_LENGTH);
      TEXT_IO.PUT_LINE("A feasible schedule found, the Earliest Start");
      TEXT_IO.PUT_LINE("Scheduling Algorithm Used. ");
      AGENDA := null;
      exit;
    exception
      when OPERATOR_SCHEDULER.MISSED_DEADLINE =>
      null;
      when OPERATOR_SCHEDULER.OVER_TIME =>
      null;
    end;

    begin
      OPERATOR_SCHEDULER.SCHEDULE_WITH_EARLIEST_DEADLINE
                                          (THE_GRAPH,AGENDA,H_B_LENGTH);
      OPERATOR_SCHEDULER.CREATE_STATIC_SCHEDULE(THE_GRAPH,AGENDA,H_B_LENGTH);
      TEXT_IO.PUT_LINE("A feasible schedule found, the Earliest Deadline ");
      TEXT_IO.PUT_LINE("Scheduling Algorithm Used. ");
      AGENDA := null;
      exit;
    exception
      when OPERATOR_SCHEDULER.MISSED_DEADLINE =>
        null;
      when OPERATOR_SCHEDULER.OVER_TIME =>
        null;
    end;
  end loop;

exception
  when FILE_PROCESSOR.CRIT_OP_LACKS_MET =>
    EXCEPTION_HANDLER.CRIT_O_L_MET(Exception_Operator);

  when FILE_PROCESSOR.MET_NOT_LESS_THAN_PERIOD =>
    EXCEPTION_HANDLER.MET_N_L_T_PERIOD(Exception_Operator);

  when FILE_PROCESSOR.MET_NOT_LESS_THAN_MRT =>
    EXCEPTION_HANDLER.MET_N_L_T_MRT(Exception_Operator);

  when FILE_PROCESSOR.MCP_NOT_LESS_THAN_MRT =>
```

```
      EXCEPTION_HANDLER.MCP_N_L_T_MRT(Exception_Operator);

    when FILE_PROCESSOR.MCP_LESS_THAN_MET =>
      EXCEPTION_HANDLER.MCP_L_T_MET(Exception_Operator);

    when FILE_PROCESSOR.MET_IS_GREATER_THAN_FINISH_WITHIN =>
      EXCEPTION_HANDLER.MET_I_G_T_FINISH_WITHIN(Exception_Operator);

    when FILE_PROCESSOR.SPORADIC_OP_LACKS_MCP =>
      EXCEPTION_HANDLER.SPORADIC_O_L_MCP(Exception_Operator);

    when FILE_PROCESSOR.SPORADIC_OP_LACKS_MRT =>
      EXCEPTION_HANDLER.SPORADIC_O_L_MRT(Exception_Operator);

    when SCHEDULE_INPUTS_LIST.BAD_VALUE =>
      EXCEPTION_HANDLER.S_I_L_BAD_VALUE;

    when DIGRAPH.V_LISTS.BAD_VALUE =>
      EXCEPTION_HANDLER.V_L_BAD_VALUE;

    when DIGRAPH.E_LISTS.BAD_VALUE =>
      EXCEPTION_HANDLER.E_L_BAD_VALUE;

    when HARMONIC_BLOCK_BUILDER.NO_BASE_BLOCK =>
      EXCEPTION_HANDLER.NO_B_BLOCK;

    when HARMONIC_BLOCK_BUILDER.NO_OPERATOR_IN_LIST =>
      EXCEPTION_HANDLER.NO_OP_IN_LIST;

    when HARMONIC_BLOCK_BUILDER.MET_NOT_LESS_THAN_PERIOD =>
      EXCEPTION_HANDLER.MET_N_L_T_PERIOD(Exception_Operator);

end STATIC_SCHEDULER;
```

LIST OF REFERENCES

1. Faulk, S., Parnas, D. *"On Synchronization in Hard Real-Time Systems"* CACM, P.274-187 (Mar, 1987)

2. Jahanian, F., Mok, A. *"Safety Analysis of Timing Properties in Real-Time Systems"* IEEETSE, SE-12, P. 890-904

3. Jensen, E., Locke, C., Tokuda, H. *"A Time-Driven Scheduling Model for Real-Time Operating Systems"* Proc. of the Real-Time Systems Symposium, San Diego, CA.IEEE, P. 112-122 [1985]

4. Luckenbaugh, G. *"The Activity List: A Design Construct for Real-Time Systems"* Master's Thesis, Department of Computer Science, UNIV of Maryland [1984]

5. Luqi, Berzins, V. *"Execution of a High Level Real-Time Language"* Proc. of the Real-Time Systems Symposium, Huntsville, Alabama (Dec, 1988)

6. Marlowe, L. *"A Schedular for Critical Timing Constrains."*, M.S. Thesis, Computer Science, Naval Postgraduate School, Monterey CA. (Dec, 1988)

7. Moitra, A. *"Analysis of Hard Real-Time Systems"* Computer Science Department, Cornell University [1985]

8. Quirk W. J. *"Verification and Validation of Real-Time Software"* SPRINGER, [1985]

9. Zave, P. *"The Operational Versus the Conventional Approach to Software Developement"* CACM, P.104-118 (Feb, 1984)

10. Stankovic J.A., Ramamritham K., Shiah P., Zhao W. *"Real-Time Scheduling Algorithms for Multiprocessors"*, COINS Technical Report 89-47, [1989]

11. Lui Sha, Lehoczky J., Rajkumar R., *"Task Scheduling in Distributed Real-Time Systems"* Department of CS, Department of Statistics, Department of Electrical and Computer Engineering, Carnegie Mellon University, [1988]

12. Sheng-Chang Cheng, Stankovic J.A., Ramamritham K. *"Scheduling Algorithms for The Real-Time Systems - A Brief Survey"*, COINS Technical Report 87-55, (June, 1987)

13. Janson D.M. *"A Static Scheduler for The Computer Aided Prototyping System"*, M.S. Thesis, Computer Science, Naval Postgraduate School, Monterey CA. (Sep, 1988)

14. O'hern J.T. *"A Conceptual Level Design for a Static Scheduler for Hard Real-Time Systems"*, M.S. Thesis, Computer Science, Naval Postgraduate School, Monterey CA. (Sep, 1988)

15. LuQi, *"Execution of Real-Time Prototypes"*, Technical Report NPS52-87-012, Nad Postgraduate School, Monterey, CA, 1987 and in *"ACM First International Workshop on Computer-Aided Software Engineering"*, Cambridge, MA, [Vol. 2: pp. 870-884] (May, 1987)

16. LuQi and Ketabchi, M., *"A Computer Aided Prototype System"*, Technical Report, NPS52-87-011, Naval Postgraduate School, Monterey, CA, 1987 and in [IEEE Software, pp. 66-72], (March, 1988)

17. LuQi, *"Handling Timing Constraints in Rapid Prototyping"*, Technical Report NPS52-88-036, Naval Postgraduate School, Monterey, CA, (Sep, 1988)

18. Moffitt, C. R., *"A Language Translator for a Computer Aided Rapid Prototyping System"*, M.S. Thesis, Computer Science, Naval Postgraduate School, Monterey CA. (Sep, 1988)

19. LuQi, *"Rapid Prototyping for Large Software System Design"*, Ph.D. Thesis, University of Minnesota, Duluth, Minnesota (May, 1986)

20. Bra,Flo,Rob71, *"Scheduling with Earliest Start and Due Date Constraints"*, Naval Research, Logistic Quarterly 18(4), (Dec, 1971)

21.Baker K.R., Sue Z.S., *"Sequencing with Due-data and Early Start Times to Minimizing Maximum Tardiness"*, Naval Research, Logistic Quarterly 21, [1971]

22.Baker K.R., Martin J.B., *"An Experimental Comparison of Scheduling Algorithms for the Single-Machine TArdiness Problem"*, Naval Research, Logistic Quarterly [1974]

23. Horn W.A., *"Some Simple Scheduling Algorithms"*, Naval Research, Logistic Quarterly 21, [1974]

24.Biyabaul S.R., Stankovic J.A., Ramamritham K., *"The Integration of Deadline and Criticalness in Hard-Real Scheduling"*, CH 2618 --7/88/0000/0152

25.Liv C.L., Laylan J.W., *"Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment"*, Journal of Association for Computing Machinery, Vol 20, No 1 [Jan. 1972: pp. 46-61]

26.Chung Jen-Yao, Liu Jane W. S., *"Algorithms for Scheduling Periodic Jobs to Minimize Average Error"*, [1987]

27. Locke C.D., Tokuda H., Jensen E.D., *"A Time-Driven Scheduling Model for Real-Time Operating Systems"*, Technical Report, Carnegie Mellon University [1985]

28. Mok A., Sutanthavibul S. *"Modeling and Scheduling of Dataflow Real-Time Systems"*, Proc. of the Real-Time Systems Symposium, San Diego, IEEE, P.178-187(Dec, 1985)

29. Booch G. *"Software Engineering with Ada"*, Menlo Park: The Benjamin/Cummings Publishing Company, 1987

30. Mok A., *"A Graph Based Computational Model for Real-Time Systems"*, Proc. of the IEEE International Conference on Parallel Processing, Pennsylvania State Univ., PA, [Aug. 20-23, 1985: pp. 619-623]

31. White J. L., *"The Development of a Rapid Prototyping Environment"*, M.S. Thesis, Computer Science, Naval Postgraduate School, Monterey CA. (Dec, 1989)

32. Palazzo F., *"Integration of the Execution Support System for Computer Aided Prototyping System"*, M.S. Thesis, Computer Science, Naval Postgraduate School, Monterey CA. (Dec, 1989)

33. Cervantes J. J., *"An optimal Static Scheduling Algorithm for Hard Real-Time Systems Specified in a Prototyping Language."*, M.S. Thesis, Computer Science, Naval Postgraduate School, Monterey CA. (Dec, 1989)

34. Eaton S.L., *"An Implementation Design of a Dynamic Scheduler for a Computer Aided Prototyping System"*, M.S. Thesis, Computer Science, Naval Postgraduate School, Monterey CA. (Mar, 1988)

INITIAL DISTRIBUTION LIST

1.	Defence Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
4.	Dr. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
5.	Luqi, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, CA 93943	5
6.	Laura J. White, Code 52Wh Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
7.	Julian Jaime Cervantes Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
8.	Murat Kilic Merkez Mah. 31. Sokak No:6, D.1 41650 Golcuk-Kocaeli/TURKEY	1
9.	Deniz Kuvvetleri Komutanligi Personel Daire Baskanligi Bakanliklar-Ankara/TURKEY	1
10.	Deniz Harp Okulu Komutanligi Kutuphanesi Tuzla-Istanbul/TURKEY	1

11.	Golcuk Tersanesi Komutanligi ARGE Sube Mudurlugu Golcuk-Kocaeli/TURKEY	1
12.	Ortadogu Teknik Universitesi Kutuphanesi Ankara/TURKEY	1
13.	Istanbul Teknik Universitesi Kutuphanesi Gumussuyu-Istanbul/TURKEY	1
14.	Bogazici Universitesi Kutuphanesi Bebek-Istanbul/TURKEY	1
15.	Mr. John White PMTC Code 1051 Point Mugu, CA 93042	1

617-584

Thesis
K41113 Kilic
c.1
Static schedulers for embedded real-time systems.

[illegible]
20 FEB 92

30454
80420

Thesis
K41113 Kilic
c.1
Static schedulers for embedded real-time systems.